1. Introduction
    1. ELEC226 Course Philosophy
    2. The MSP430F16x Deluxe Development Board
    3. The MSP430F16x Lite Development Board
2. Background
    1. What is a Microcontroller?
    2. Binary and Hexadecimal Notation
    3. What is digital?
    4. How to Read Datasheets
    5. CPU Registers in the MSP430
3. Labs
    1. Lab1 - Lab Equipment
        1. Using a Basic Function Generator
        2. Using an Oscilloscope
        3. Lab 1: Using the Lab Hardware and Reading Datasheets
    2. Lab 2 - Programming the MSP430
        1. What is a program?
        2. Introduction to CrossStudio MSP430 IDE
        3. Introduction to Programming the MSP430
        4. Setting Breakpoints in Crossworks
        5. Lab 2: C and Macros with Texas Instruments' MSP430
    3. Lab 3 - Assembly Language Programming
        1. Introduction to Assembly Language
        2. Structure of an Assembly Program
        3. Lab 3: Introduction to Assembly Language
    4. Lab 4 - Clocking
        1. What is a digital clock?
        2. Clock System on the MSP430
        3. Lab 4: Clocking on MSP430

5. Lab 5 - Interrupts
    1. Interrupts
    2. Lab 5: Interrupts
6. Lab 6 - Timers
    1. Timers on the MSP430
    2. Watchdog Timer
    3. Lab 6: Timers on the MSP430
7. Lab 7 - Mixed Signal Processing
    1. Introduction to Sampling
    2. Analog-to-Digital Converter on the MSP430
    3. Digital-to-Analog Converter on TI MSP430
    4. Lab 7: ADC, DAC, and Mixed Signals
8. Lab 8 - DMA & RS232
    1. Real Time
    2. What is Direct Memory Access (DMA)?
    3. Lab 8: DMA + RS232
9. Lab 9 - Low Power and Optimization
    1. Memory Conservation
    2. Improving Speed and Performance
    3. Reducing Power Consumption
    4. Lab 9: Optimization and Low Power Modes
10. Lab 10 - FIR Filtering
    1. FIR Filters
    2. Lab 10: Implementing an FIR filter

ELEC226 Course Philosophy

Course Overview

Note: Some of the terms in this module may have no meaning for you now. That is fine. By the end of the course, you will understand them all.

This is a course about microcontrollers and embedded systems, specifically the Texas Instruments MSP430F169 microcontroller. This course is a hands-on laboratory experience, and is primarily intended for an audience freshman and sophomore undergraduate students, though anyone who is interested in knowing about this subject matter is encouraged to take it. No prerequisites are required, and a best effort is made to teach you (the student) as much as possible about microcontroller systems. Therefore, I try to make no assumptions about your previous experience with digital logic, programming computing systems of any sort, and detailed understanding of hardware.

By the end of this course, you will have learned many things related to designing a system with a microcontroller. The emphasis is primarily on the design of software for microcontroller systems, and as such, this is not a hardware design course. However, following this course you will have enough detailed understanding of microcontroller architecture and operation such that a follow-on hardware design course would allow you to sucessfully produce your own hardware/software system. Among the main concepts you will learn in this course are:

ELEC226 Course Concepts

- What a microcontroller is, what type of applications they are used for, and how to use one.
- The basics of structured programming using the C language and, to a lesser extent, assembly language.

- How to compile, execute and debug your programs on a microcontroller.
- Understand the architecture of microcontrollers in general and the specific architecture of the MSP430F169, including how to interface with the on-chip peripherals such as the ADC and DAC.
- Interfacing with off-chip peripheral hardware, such as expansion memory.
- Microcontroller programming techniques: using timers, clock and power management, task scheduling, etc...
- Simple digital signal processing.

Course Structure

The course structure will consist of one weekly lecture to discuss topics in microcontroller architecture and embedded systems design. In addition, there will be weekly hands-on lab sessions. At the end of the semester, the students will have to complete a project which will combine many of the skills they have learned during the course. There will be occasional in-class quizzes to reinforce the material learned in lab.

Grading

The grading system is shown in the table below. Most of the grade is determined byt the sucessful completion of the lab work and by the end-of-semester project.

Item	Percentage
Weekly Labs	50%
Final Project	30%

Item	Percentage
Quizes	20%

ELEC226 Grading

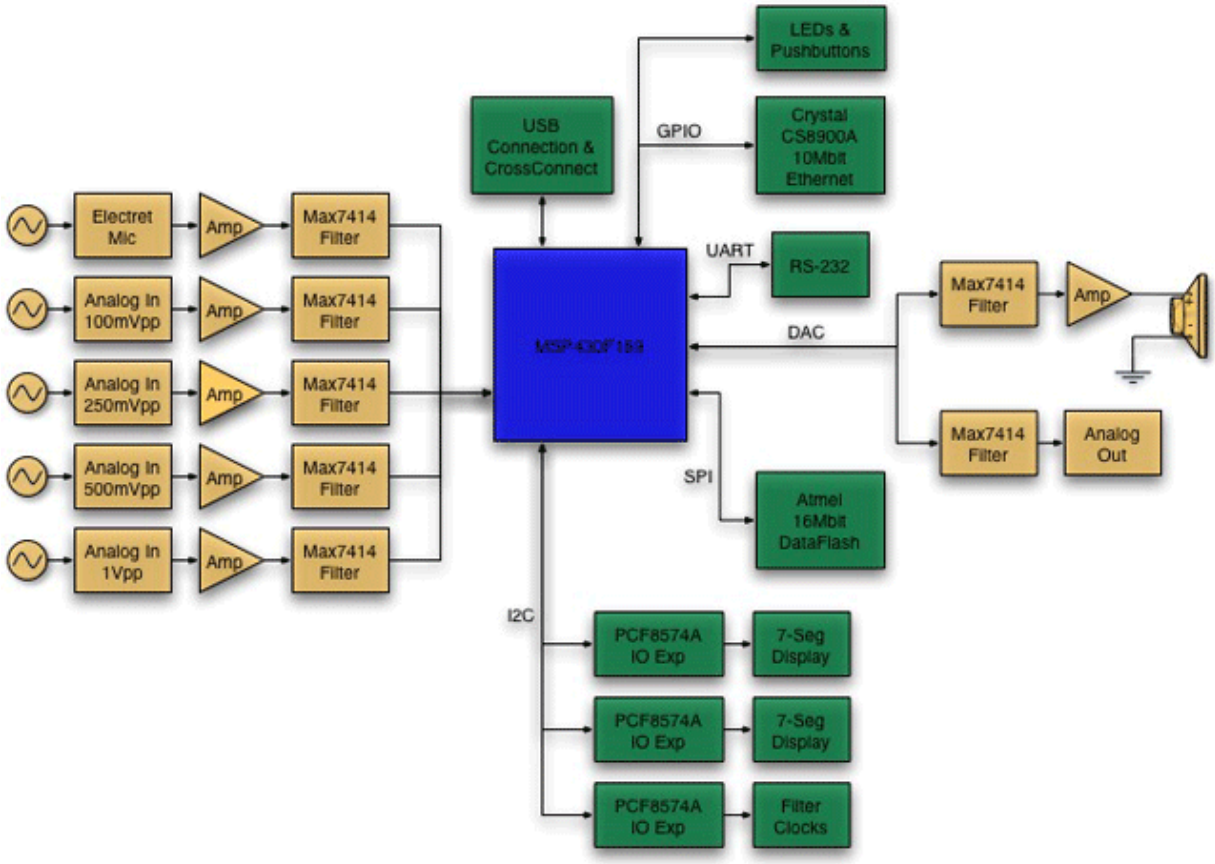
The MSP430F16x Deluxe Development Board

Overview

This module describes a full-featured development board for the Texas Instruments MSP430F16x series of mixed-signal microprocessors. This hardware system was designed to allow students or engineers to fully excercise and explore the capabilities of the MSP430 microcontroler.

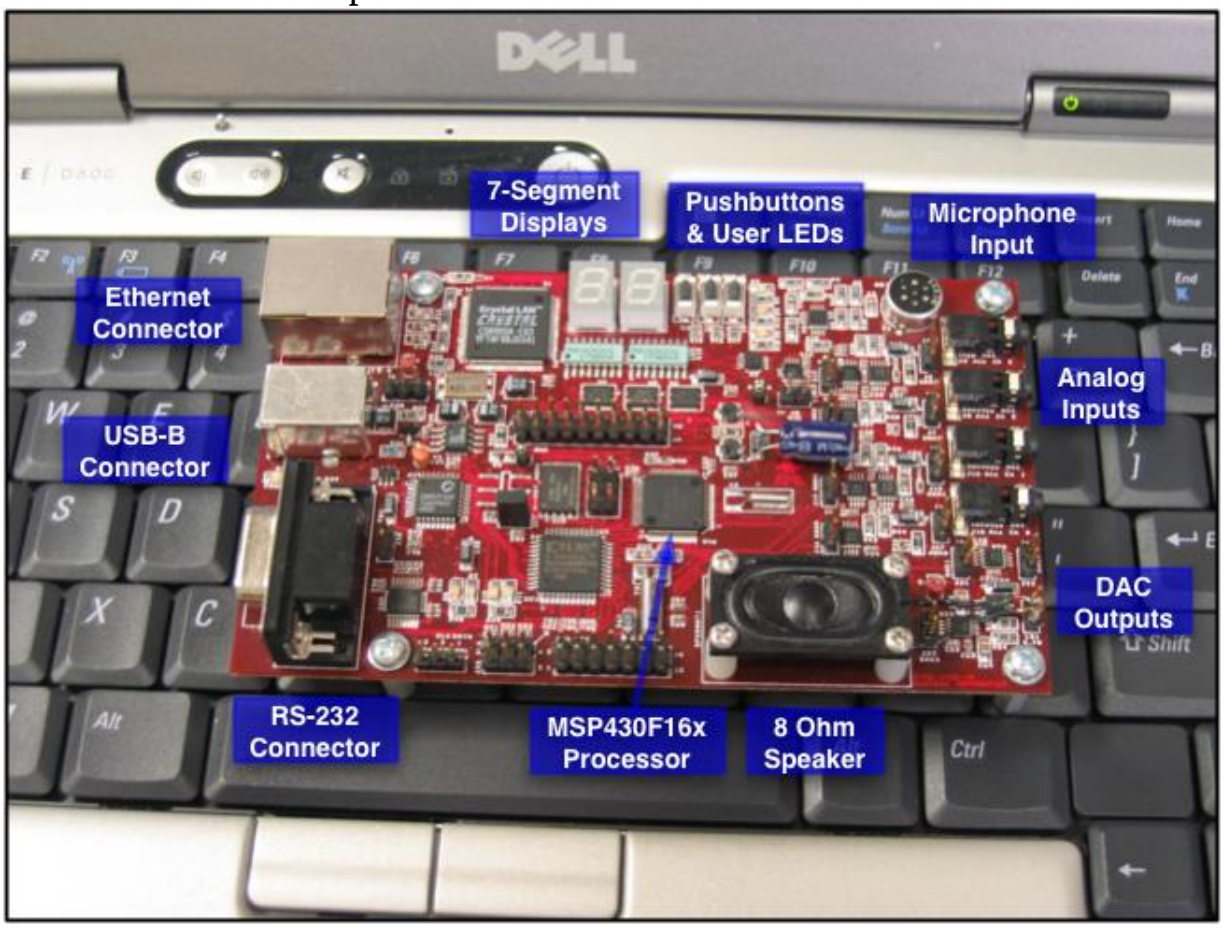
This module is meant to serve as a general technical overview and instruction manual for the development board. The two pictures below show a general block diagram of the development system and a picture of the board that highlights some of the major peripherals.

MSP430F16x Development Board Block Diagram



A simplified system block diagram for the MSP530F16x Deluxe Development Board.

MSP430F16x Development Board



A Photo MSP530F16x Deluxe Development Board with Major Peripherals Highlighted.

System Features Overview

The MSP430F16x deluxe development board was designed to offer acces to a broad range of the peripheral devices available on the MSP430, thus allowing the testing and evaluation of the full capabilities of the MSP430 line of processors.

MSP430F16x Deluxe Development Board Features List

- Digital Section

    - Texas Instruments MSP430F16x MSP430F16x (1611 or 169) 16-bit microcontroller.
        - 48KB of flash code emory and 10KB of SRAM data with the MSP430F1611.
        - 60KB of flash code emory and 2KB of SRAM data with the MSP430F169.
    - Embedded USB CrossConnect JTAG Emulator.
    - RS-232 Serial PC Interface
    - 2MByte SPI DataFlash
    - 10B-T Ethernet
    - Up to 16 User-Accesible Digital I/Os
    - Dual 7-Segment Displays
    - 2 Debounced User Pushbuttons with Interrupt Capability
    - 3 User LEDs
- Analog Section
    - Electret Microphone Input
    - 4 Amplified Analog Inputs: 100mVpp, 250mVpp, 500mVpp and 1Vpp (amplification can be bypassed for direct acess to the ADC input)
    - Programmable pre-ADC anti-alias filtering using the MAX7414, a 5th order lowpass Butterworth filter with a tunable f_c from 1Hz to 15kHz.
    - 2 DAC outputs, one with 0.7W amplification to an 8 Ohm Speaker (amplification can be bypassed for direct access to the DAC output)
    - Programmable post-DAC filtering using the MAX7414
    - MSP control over the following: pre-ADC and post-DAC filter shutdown, pre-ADC and post-DAC filter f_c, and power amplifier shutdown.
    - System Current Measurement

Supplemental Documents

MSP430F16x Deluxe Development Board Schematics

- ELEC226 Schematics: The schematics contain all pinouts, header information, and connections between devices on the ELEC226 board.

MSP430F16x - Mixed Signal Microcontroller

- MSP430F169: Mixed Signal Microcontroller Datasheet
- MSP430x1xx Family User's Guide: The user's guide discusses modules and peripherals of the MSP430x1xx family of devices. This may be the most useful reference for programming the MSP430.

Analog I/O

- Analog, Lowpass Filter (MAX7414) Datasheet
- Audio Power Amplifier (TPA721) Datasheet
- Op-Amp MicroAmplifier (OPA2244) Datasheet

JTAG & CrossConnect

- Xilinx PLD (XCR3032XL) Datasheet
- USB interface MCU (C8051F320) Datasheet
- 500mA Linear Regulator (MAX604CSA) Datasheet

RS-232

- RS-232 Transceiver (MAX3221CUE) Datasheet

16Mbit DataFlash

- 16Mbit DataFlash (AT45DB161B) Datasheet

Ethernet

- Crystal LAN Embedded Ethernet Controller (CS8900A) Datasheet

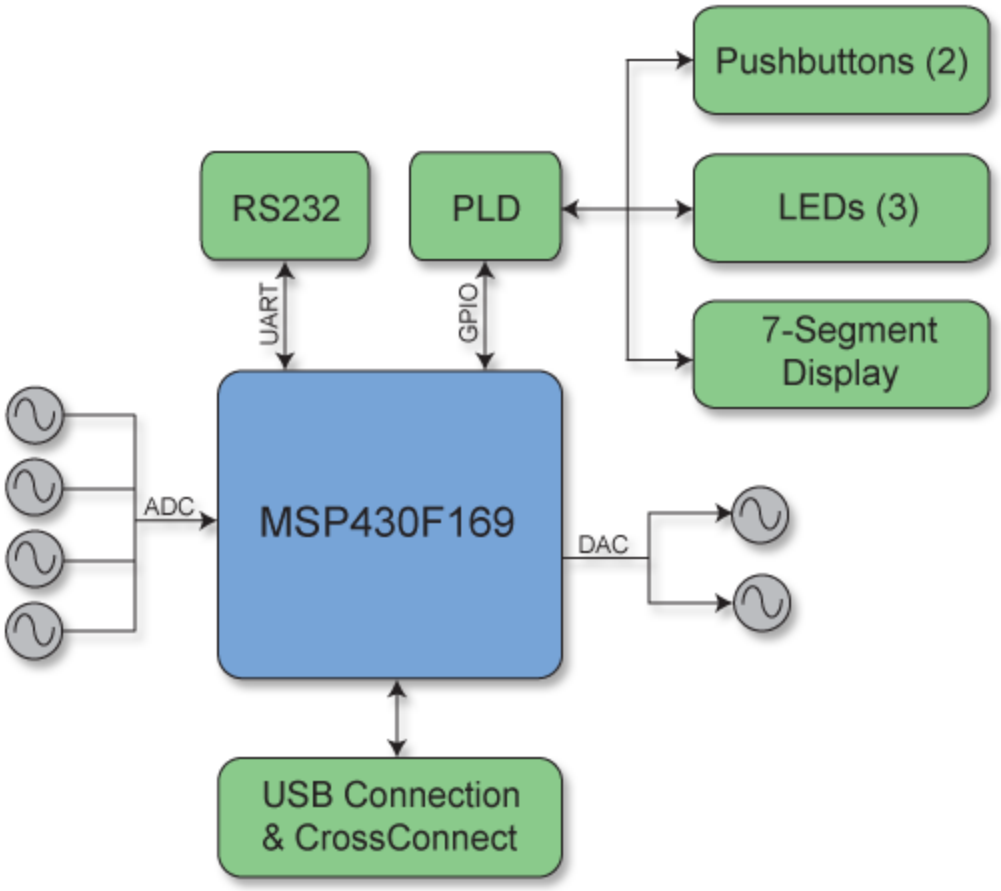
The MSP430F16x Lite Development Board

Overview

This module describes a full-featured development board for the Texas Instruments MSP430F16x series of mixed-signal microprocessors. This hardware system was designed to allow students or engineers to fully exercise and explore the capabilities of the MSP430 microcontroller.

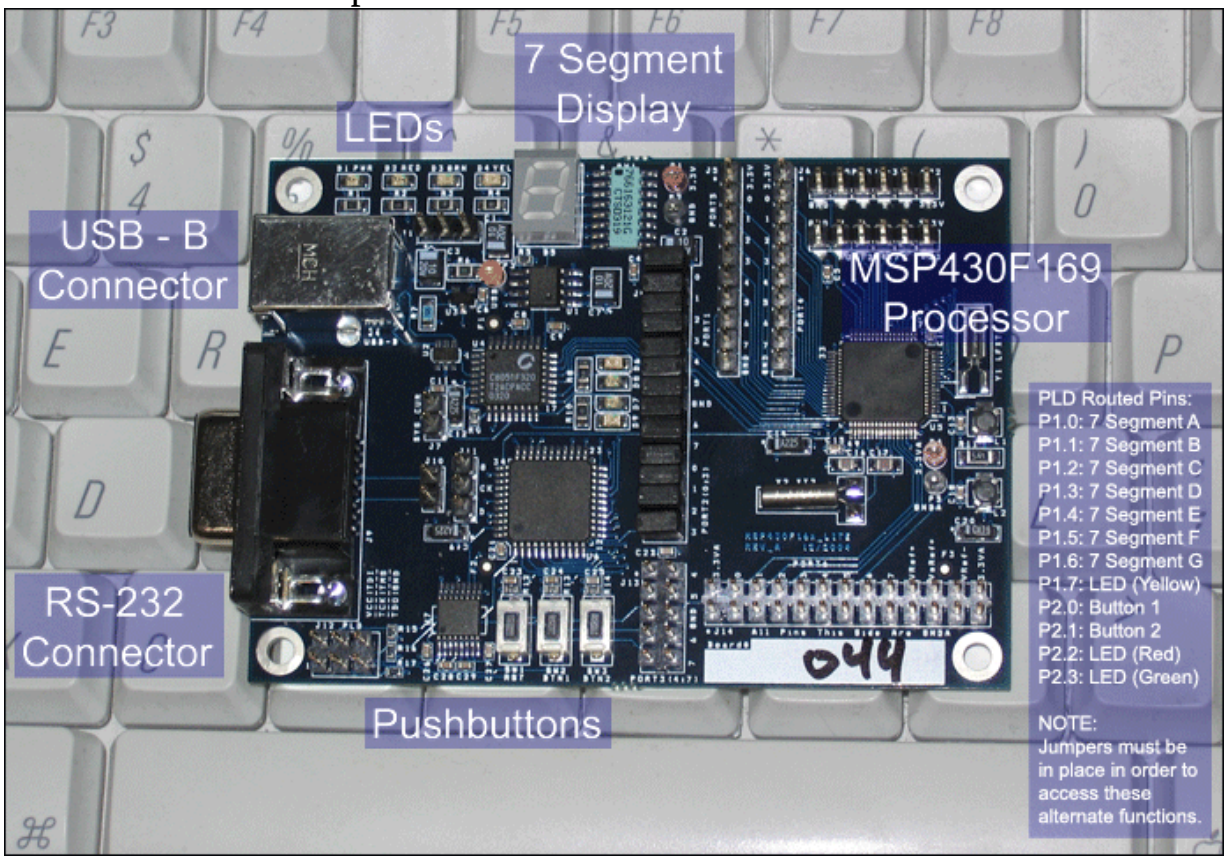
This module is meant to serve as a general technical overview and instruction manual for the development board. The two pictures below show a general block diagram of the development system and a picture of the board that highlights some of the major peripherals.

MSP430F16x Lite Development Board Block Diagram



A simplified system block diagram for the MSP430F16x Lite Development Board.

MSP430F16x Development Board



A photo of the MSP430F16x Lite Development Board with major peripherals highlighted. You may verify the Lite Board Pinout here.

System Features Overview

The MSP430F16x Lite development board is a streamlined version of the MSP430F16x Deluxe Development Board. It allows for testing and evaluation of the full capabilities of the MSP430 line of processors, yet has been reduced in cost and power consumption.

MSP430F16x Lite Development Board Features List

- Digital Section

- Texas Instruments MSP430F16x MSP430F16x (1611 or 169) 16-bit microcontroller.
    - 48KB of flash code memory and 10KB of SRAM data with the MSP430F1611.
    - 60KB of flash code memory and 2KB of SRAM data with the MSP430F169.
- Embedded USB CrossConnect JTAG Emulator.
- RS-232 Serial PC Interface
- Up to 16 User-Accessible Digital I/Os
- Single 7-Segment Display
- 2 Debounced User Pushbuttons with Interrupt Capability
- 3 User LEDs
- System Current Measurement

Supplemental Documents

MSP430F16x Lite Development Board Schematics

- Board Schematics: The schematics contain all pinouts, header information, and connections between devices on the ELEC226 board.

MSP430F16x - Mixed Signal Microcontroller

- MSP430F169: Mixed Signal Microcontroller Datasheet
- MSP430x1xx Family User's Guide: The User's Guide discusses modules and peripherals of the MSP430x1xx family of devices. This may be the most useful reference for programming the MSP430.

JTAG & CrossConnect

- Xilinx PLD (XCR3032XL): Some signals are routed through the PLD in order to change the mapping of the connected peripheral. The

current mapping is shown in the picture above, however, if they are not accurate, you may verify the Lite Board Pinout here.
- USB interface MCU (C8051F320) Datasheet
- 500mA Linear Regulator (MAX604CSA) Datasheet

RS-232

- RS-232 Transceiver (MAX3221CUE) Datasheet

What is a Microcontroller?

Consider the following set of words: microprocessor, microcontroller, processor, digital signal processor, mixed signal processor. In one sense, they are all the same thing - an **ASIC** that fetches and executes instructions based on input from some user program. These devices do not have a fixed function, but rather are controlled by software. Each of them share certain architecural features that have been developed since Intel created the first "microprocessor" in 1971.

Note: Intel's web site contains an interesting history of the microprocessor.

In the three decades since the invention of the first microprocessor, there has been tremendous development and innovation in this field of engineering. Each of the terms used at the start of this section are correct names for a microprocessor, but they all have different application spaces and fetures. This still leaves us with the question, "What is a microcontroller?"

In truth, this is a bit difficut to pin down, due to the ever-evolving nature of the semiconductor industry. Consider that what we would call today's average microcontroller is oders of magnitude more powerful than the computer used in the Apollo Lunar Module. We can make some generalizations, however, that can help us characterize a microcontroller. Typically, these devices reside at what is the "low-end" of computing. This does not, however, mean that microcontrollers are useless. For **embedded systems** design, microcontrollers are usually an ideal choice. The following list shows some qualities that define all microprocessors, along with how they specifically define microcontrollers.

Some Common Processor Charateristics

- **Cost**: The cost of the part. Microcontrollers are usually very cheap, sometimes even less than $1 each.
- **Speed**: The frequency (speed) of the system clock, often stated in **megahertz** (MHz) or **gigahertz** (GHz). Microcontrollers are typically slow, less than 20MHz.
- **Power**: The power dissapation of a system, measured in **Watts**. Microcontrollers are almost always "low-power."
- **Bits**: This usually means the number of bits that can be processed at one time by each instruction (e.g. 8-bit, 16-bit, 32-bit, etc...). Microcontrollers are almost always 8-bit or maybe 16-bit devices.
- **Memory**: Most processors have some amount of storage on the device for data and program instructions. In computing systems, memory is often hierarchical, so on-chip memory can serve different purposes. Microcontrollers typically have a limited amount of memory, less than 1MByte total.
- **Input/Output (I/O)**: All processors have some means of getting data in and out of the chip. In the physical sense, this relates to metal pins on the part package which are used to connect to other circuitry in the system. Microcontrollers usually have just a few I/O pins, as few as 8 or as many as 100. Larger processors, such as the ones founds in a typical PC, will typically have hundreds of pins.

Note: Some of the characteristics listed above are interdependent. For example, the greater number of pins a particular part has, the more complicated the packaging will need to be, which will probably cost more.

The figure below shows a photo of a modern microcontroller that meets all of these criteria.
Texas Instruments MSP430F169 16-bit Microcontroller

The TI MSP430F169 meets the criteria we listed above and is a good example of a modern microcontroller.

As a last excercise, it is useful to compare different types of processors to see the tremendous amount of variety available. The following table shows a selection of modern processors and some numbers related to the features listed above. It is clear that there are tradeoffs to be made in choosing the right part for the design at hand, and part of being a good engineer is being able to do this well.

Processor	Manufacturer	Cost	Speed	Power	Bits	On-Chip Memory	Off-Chip Memory
MSP430F169	Texas Instruments	$1 - $10	8 MHz	~5 mW	16-bit	1 - 62 KByte	N/A
Pentium II	Intel	~$65	333 MHz	~25 W	32-bit	548 KByte	4 GByte
TMS320C6416	Texas Instruments	~$300	700 MHz	~1.5 W	16/32-bit	1 MByte	1.28 GByte
PowerPC 970	IBM	N/A	1.8 GHz	~42 W	64-bit	608 KByte	8 GByte

Comparisson of Modern Processors

Glossary

ASIC
: Application Specific Integrated Circuit

embedded system
: A self contained electronic control system, generally with limited user input/output.

> **Note:**A traffic light control system is a good example of an embedded system. The system is self-contained and controls the flow of vehicles at an intersection. Users (pedestrians, bicyclists and car drivers) interact with the system in a limited manner. There may be sensors that detect the presence of vehicles and buttons

for pedestrians to push when they want to cross the intersection. The traffic lights output the "state" of the system and inform the users of the actions they must take.

Note: The best desinged embedded systems are those that are unobtrusive to the user. That is, they require little user interaction and, for the most part, are "invisible."

Hertz
: A unit of measurement for frequency, abbreviated Hz, denoting the number of cycles per second. 1 Hz = 1 cycle/second. For a more detailed explanation of frequency, see the following module.

Example:
1 megahertz (MHz) = 1,000,000 cycles/second.

Example:
1 gigahertz (GHz) = 1,000,000,000 cycles/second.

Watt
: A unit of measurement for power, abbreviated W, denoting the amount of energy (in joules) per second. 1 W = 1 joule/second.

Binary and Hexadecimal Notation

Because of the nature of the digital systems, it is necessary to be able to represent numbers as being composed of only 1's and 0's. Ordinarily we represent numbers using the characters 0-9, and this notation is called **base 10** or **decimal** notation. Using only the characters 1 and 0 is called **binary** notation or **base 2** (because there are only 2 characters to represent the number instead of 10). Converting integers between these two systems is easy. In base 10 each decimal place represents the number of a certain power of 10. Thus the one's place(10^0), 10's place(10^1), 100's place (10^2) etc. In base 2 each place represents a corresponding power of 2.

To convert a base 10 number into its base 2 form, begin at the 2's place of the largest power of two smaller than the base 10 number you are converting. This will be the highest 1 digit of the base two number. Now see if the next smaller power of 2 is larger than the reminder of your base 10 number. If it is the next place in the base two number is a 0 if its smaller, subtract the power of two from the base 10 number and put a 1 in the next place. Repeat this until you have reached the 2^0 place. To convert back, go through each power of two place in the binary number and multiply it by the corresponding power of two. Sum these products to get the decimal version of the binary number.

Example:
steps in converting to base 2

1. $721 - 512 = 209$ so the first bit is 1×2^9
2. $209 < 256$ so the second bit is 0×2^8
3. $209 - 128 = 81$so the third bit is 0×2^7
4. $81 - 64 = 17$ so the fourth bit is 1×2^6
5. $17 < 32$ so the fifth bit is 0×2^5
6. $17 - 16 = 1$ so the sixth bit is 1×2^4
7. $1 < 8$ so the seventh bit is 0×2^3
8. $1 < 4$ so the eigth bit is 0×2^2
9. $1 < 2$ so the ninth bit is 0×2^1
10. $1 - 1 = 0$ so the tenth bit is 1×2^0

11. thus the conversion:$721 = 1011010001$

This method works backwards also, starting from 1011010001 and expanding each digit by its appropriate exponent.

Exercise:

Problem: What is 293 in binary? What is 1110001 in decimal?

Solution:

$293 = 100100101$ and $1110001 = 113$

Hexadecimal is another numerical convention that is really **base 16**. It uses the characters 0-9 for its first 10 numbers and the letters A-F to represent 10 through 15. Conversion between it and base 10 numbers proceeds the same as base 2 substituting powers of 16 for powers of 2. However, the important use of hexadecimal numbers is as an abbreviation for binary; because binary representation of large numbers becomes quite long. When programming, hexadecimal numbers should be prefaced with 0x to indicate that they are hexadecimal numbers rather than variable names etc. Thus 14 is a decimal number in C, and 0x14 is a hexadecimal number (actually equal to 20 in decimal). Below is a the list of expansions for hexadecimal to binary to decimal.

binary to hexadecimal equivalence

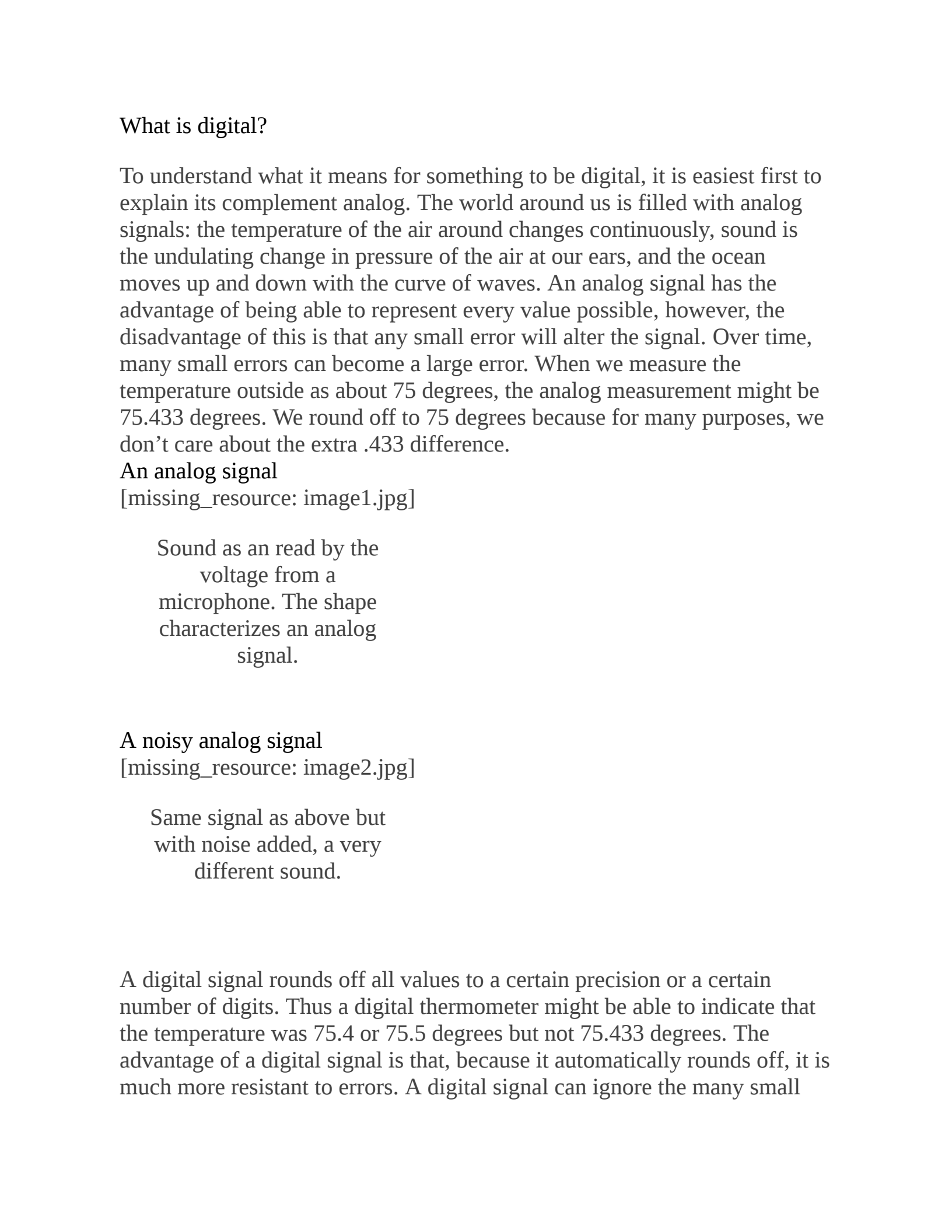
- $0000 = 0 = 0$
- $0001 = 1 = 1$
- $0010 = 2 = 2$
- $0011 = 3 = 3$
- $0100 = 4 = 4$
- $0101 = 5 = 5$
- $0110 = 6 = 6$
- $0111 = 7 = 7$
- $1000 = 8 = 8$

- $1001 = 9 = 9$
- $1010 = \mathrm{A} = 10$
- $1011 = \mathrm{B} = 11$
- $1100 = \mathrm{C} = 12$
- $1101 = \mathrm{D} = 13$
- $1110 = \mathrm{E} = 14$
- $1111 = \mathrm{F} = 15$

What is digital?

To understand what it means for something to be digital, it is easiest first to explain its complement analog. The world around us is filled with analog signals: the temperature of the air around changes continuously, sound is the undulating change in pressure of the air at our ears, and the ocean moves up and down with the curve of waves. An analog signal has the advantage of being able to represent every value possible, however, the disadvantage of this is that any small error will alter the signal. Over time, many small errors can become a large error. When we measure the temperature outside as about 75 degrees, the analog measurement might be 75.433 degrees. We round off to 75 degrees because for many purposes, we don't care about the extra .433 difference.

An analog signal


[missing_resource: image1.jpg]

Sound as an read by the voltage from a microphone. The shape characterizes an analog signal.

A noisy analog signal

[missing_resource: image2.jpg]

Same signal as above but with noise added, a very different sound.

A digital signal rounds off all values to a certain precision or a certain number of digits. Thus a digital thermometer might be able to indicate that the temperature was 75.4 or 75.5 degrees but not 75.433 degrees. The advantage of a digital signal is that, because it automatically rounds off, it is much more resistant to errors. A digital signal can ignore the many small

errors which, over time, would become a large error in an analog signal. Errors that appear in a signal are called noise. As long as the volume of the noise remains small relative to the difference between two levels of a digital signal, the noise will not affect the digital signal at all. Any amount of noise affects an analog signal.

A digital signal

[missing_resource: image3.jpg]

A digital signal and its logical interpretation.

A noisy digital signal

[missing_resource: image4.jpg]

A noisy digital signal can still maintain the same logical interpretation.

In computers, the digital signal is either on or off. If a signal's voltage level is close to 0V then it's interpreted as off or 0. If the signal is close to the operating voltage of the device, say 3.3V, then the signal is interpreted as on or 1. Voltages in between are rounded off to on or off, but most devices are designed to keep the signals at on of the two extremes. While the computer operates as if the signal were either a 1 or a 0, the underlying voltage is still an analog value like .121V or 3.1.V

How to Read Datasheets

For every electronic component or series of components, the manufacturer or designer produces a data sheet. In its early stages, a data sheet might be the specifications the designer works from; but, by the time the device is released, the data sheet is the essential piece of information that describes exactly what the component does. Everything from the smallest resistor to the most elaborate processor needs a datasheet. Datasheets focus on electrical properties and the pin functions of the device; usually the inner workings of the device are not discussed. This is partly to make industrial espionage more difficult, and also because the user should not need to know the internal workings of the device. In practice, if you find that you need to know how a particular product works internally, you can often call the manufacturer and find out what you need to know.

In addition to datasheets, devices with complex configurations or applications may have related documents to help the designer work with their products. These are called “application notes,” “user’s guides,” “designer’s guides,” “package drawings,” etc. These documents are usually just as necessary as the datasheet. In general, it is best to get the datasheet directly from the company website relatively often because occasionally there is errata or new information to be found in the datasheets. Datasheets are invariably covered with legal disclaimers as to the accuracy, permanency, and utility of the document. Below are some of the kinds of things you might find in a datasheet and explanations of their usual meaning.

Official name of the part or series, part numbers and part number variations and manufacturer release date of the datasheet. Part number variations usually indicate alternative packaging or temperature tolerance. For component families, a chart might be provided to helpfully graph all of the family members. Price is usually not indicated on a datasheet.

An overview of the parts purpose and features is usually included near the beginning. This is what you scan to see if the part is what you think it is.

The electrical operating characteristics section of a datasheet indicates the minimum and maximum voltage and current parameters for the chip as a

whole and for individual pins. Power requirements for the chip as a whole are essential. The power supply circuitry of the device must be able to support all of the components. Operating frequencies of clocks or information are often indicated here. Pin electrical characteristics are important when using the pin to drive larger loads. Lower power integrated circuits are not always capable of driving an LED, for example. Noise tolerance of the power supply, or noise created by the component might also be found here. Capacitance, inductance, and resistance caused by the component might also be found here. These are especially important for high-speed circuit analysis.

The datasheet should also note the tolerances of the device. While the above operating characteristics indicated what conditions are needed for the device to operate as promised, the tolerances indicate the maximum and minimum conditions the component can handle without permanent damage. Both the operating conditions and the tolerances should indicate the nature of the testing experiments. Voltage, temperature, moisture, air pressure, ultraviolet radiation, and physical stress are possible tolerance conditions.

The arrangement and name of each pin on the chip is a necessity on any integrated circuit (IC) datasheet. The diagram should specify whether the diagram is from a point of view above or below the chip and list the pins by name or number. Pin functional descriptions should accompany the pin map diagram to explain the basic purpose of each pin. If this short description is insufficient, a more elaborate explanation is usually included later in the datasheet. Often the short descriptions may not be 100% clear to a novice data sheet reader because of abbreviations or conventions. If the data sheet doesn’t fully explain what the short description means, the term is probably common enough to be found elsewhere on the Internet.

A block diagram of architecture might be included for more complex devices. Other internal descriptions might be provided, but usually the description is limited to the parts of the system that the user can access.

Waveforms of input or outputs are common. This is especially true for explaining bus operation and data formats. Timing diagrams and information are also essential for finding interoperable devices. Just

because two components use the same bus protocol does not always mean they can talk to each other. Checking this information is always a good idea.

Graphs of I/V curves, noise profiles, input response, performance descriptions are very common. For system/control behavior this can be useful, but the testing conditions are not always entirely clear. This kind of information is the basis of the analysis many engineers do, but the graphs are rarely a good substitute for prototyping.

Many kinds of components only work if accompanied by necessary passive components. Usually these systems provide an “example” configuration that will produce a known behavior. Examples are very useful if you have the same needs the example configuration claims to meet, but the datasheet should also include the formulas and explanation necessary to pick your own accompanying components.

Distribution information and manufacturing or assembly advice might also be found in a datasheet. For example, a crystal clock might specify that it should be soldered to the circuit board for no more than 10 seconds at 400 degrees. For commercial design this kind of information is useful, as adhering to such recommendations improves yield.

Mechanical drawing and footprints are the last piece of essential information a datasheet includes. This will be a drawing of the physical form of the device, with measurements specified in metric and American units. For designing a board, it is very important to understand the drawings because incorrectly interpreting the relationship among the pins will waste an entire revision of the board. The usual way of describing the dimensions is to specify a pin width and spacing precisely, but to give broader tolerances on the exact length and width of the overall device. What this means is that the process of manufacturing the ICs tries to control the width and spacing of the pins, but everything else is somewhat flexible.

CPU Registers in the MSP430

The MSP430 has 16 CPU registers. Of these 16, the upper 12 are general purpose 16 bit registers (R4-R15). The lower four are:

- R0 Program Counter(PC) – This register controls the next instruction to be executed by the MSP core. In general, this register is incremented automatically during execution. It can be used as a source in operations normally.
- R1 Stack pointer (SP) – The stack pointer is used to keep track of previous execution modes and to return from interrupts. Can be read as a normal register.
- R2 Status Register (SR) – The status register can be written to change the operating mode of the MSP as specified in the User's Guide. When read it can act as a constant generator. Depending on the instruction code options this register will be read as: a normal register, 0x0000, 0x0004, or 0x0008 depending on the As bits.
- R3 Constant Generator II – This register cannot be written to, and when read produces: 0x0000, 0x0001, 0x0002, or 0xffff depending on the As bits.

The rest of the registers on the MSP430 behave as if they were memory. In most cases, these special purpose registers can be read and written to normally; but they affect the behavior of their respective systems.

Using a Basic Function Generator

What is a function generator?

A function generator is a device that can produce various patterns of voltage at a variety of frequencies and amplitudes. It is used to test the response of circuits to common input signals. The electrical leads from the device are attached to the ground and signal input terminals of the device under test.

Features and controls

Most function generators allow the user to choose the shape of the output from a small number of options.

- Square wave - The signal goes directly from high to low voltage.
- Sine wave - The signal curves like a sinusoid from high to low voltage.
- Triangle wave - The signal goes from high to low voltage at a fixed rate.

The amplitude control on a function generator varies the voltage difference between the high and low voltage of the output signal.

The direct current (DC) offset control on a function generator varies the average voltage of a signal relative to the ground.

The frequency control of a function generator controls the rate at which output signal oscillates. On some function generators, the frequency control is a combination of different controls. One set of controls chooses the broad frequency range (order of magnitude) and the other selects the precise frequency. This allows the function generator to handle the enormous variation in frequency scale needed for signals.

The duty cycle of a signal refers to the ratio of high voltage to low voltage time in a square wave signal.

How to use a function generator

After powering on the function generator, the output signal needs to be configured to the desired shape. Typically, this means connecting the signal and ground leads to an oscilloscope to check the controls. Adjust the function generator until the output signal is correct, then attach the signal and ground leads from the function generator to the input and ground of the device under test. For some applications, the negative lead of the function generator should attach to a negative input of the device, but usually attaching to ground is sufficient.

Using an Oscilloscope

What is an oscilloscope?

An oscilloscope is a device that measures and displays voltages as a time versus voltage graph. The voltage difference between the positive and negative probe leads is measured, buffered, and displayed on the screen as a continuous curve. Because the quality of the scope affects how finely it can resolve changes in time and voltage, very sensitive or small signals may not be measurable with all oscilloscopes. For most educational purposes, however, an average oscilloscope will display the signal well enough.

Oscilloscopes are generally used to see if a circuit is performing as expected, but oscilloscopes are also useful for comparing different signals to each other. Comparisons and absolute measurements are made by comparing the signal to the graph on the display.

Oscilloscope Controls

The most common oscilloscope controls are for amplitude, frequency, triggering, and signal comparison.

The amplitude adjustment of an oscilloscope controls how tall a given voltage will appear on the screen. Usually the screen will be marked off with horizontal lines to indicate the signal's voltage. The absolute voltage per horizontal line is adjustable. Thus, if the amplitude is set to 1V/ then each block a signal is tall is 1 V. The purpose of this adjustment is that you can see a very large or a very small signal on the same screen.

The time adjustment of an oscilloscope is how much time will a certain distance across the screen represent. The vertical lines most oscilloscopes have are the standard distance and the knob or screen indicator will describe how time that distance represents. The purpose of this adjustment is to be able to see a very quickly changing or a slowly changing signal on the same screen.

Triggering refers to the means by which the oscilloscope selects the exact moment to display on the screen. Because electrical signals often change far faster than a human being could observe them, it is necessary to only display a small sampling of the signal. If a signal has a repeating pattern, then the pattern will be repeatedly shown on the screen so that it can be viewed continuously. If the signal does not have a regular pattern then the oscilloscope will have trouble choosing the best moment to refresh the display. In this case, you will need to set up a trigger so that the oscilloscope can freeze the screen at the right time. Trigger controls allow for conditions such as an upward edge of a signal, a voltage glitch, or a voltage threshold. Trigger controls can also allow for the display refresh to occur after a time delay from the time of the trigger event. You should adjust thee triggering of an oscilloscope if the signal seems to be sliding left and right on the display or changing too fast to be seen.

Because many oscilloscopes allow multiple signals to be compared at the same time, it is necessary to have controls to handle the display of these signals. The most basic control is the ability to turn off the channels that are not in use to avoid useless cluttering of the display screen. Oscilloscopes also allow for two different channels to be compared by displaying the additive difference between them instead of the signals individually. Usually different signals can also be displayed with different amplitude adjustments, but they require the same frequency adjustment. Finally, if special triggering conditions are used, the trigger system will ask for which channel should trigger the display.

Hazards to Avoid

Triggering is probably the trickiest skill to acquire. Generally, adjusting the voltage level for the default trigger will fix most triggering problems on analog signals. For more complicated observations, the most useful tool in triggering is a time delay. While exact moment of the signal that is interesting to you may not be easy to trigger on, often this moment occurs at a fixed time offset from an event that is easy to trigger on. Try finding a signal transition or voltage threshold that stabilizes the display, then adjusting the time delay to observe your signal.

It is important to ensure that both probes the positive and negative leads of the probe are attached to the device. Usually the negative lead can attach to the ground plane of the device,

Oscilloscope probes must be attached to the signal being measured and to the ground plane of the circuit too. Noise is introduced if both probes are not properly attached to the device.

It is also important to note that the probe itself has a 1M ohm resistance. This means that from most digital circuits, the probe will not draw any significant power. However if you are trying to measure voltage from a node between two 10M ohm resistors, it would drop the voltage by about half! The lesson is that you will sometimes need to be aware of how the probe affects what you are measuring. Finally the capacitance and inductance of the probe itself can affect finely tuned analog signals. Very high speed signals (approaching 100MHz) can be very difficult to measure with an oscilloscope.

Lab 1: Using the Lab Hardware and Reading Datasheets

Exercise:

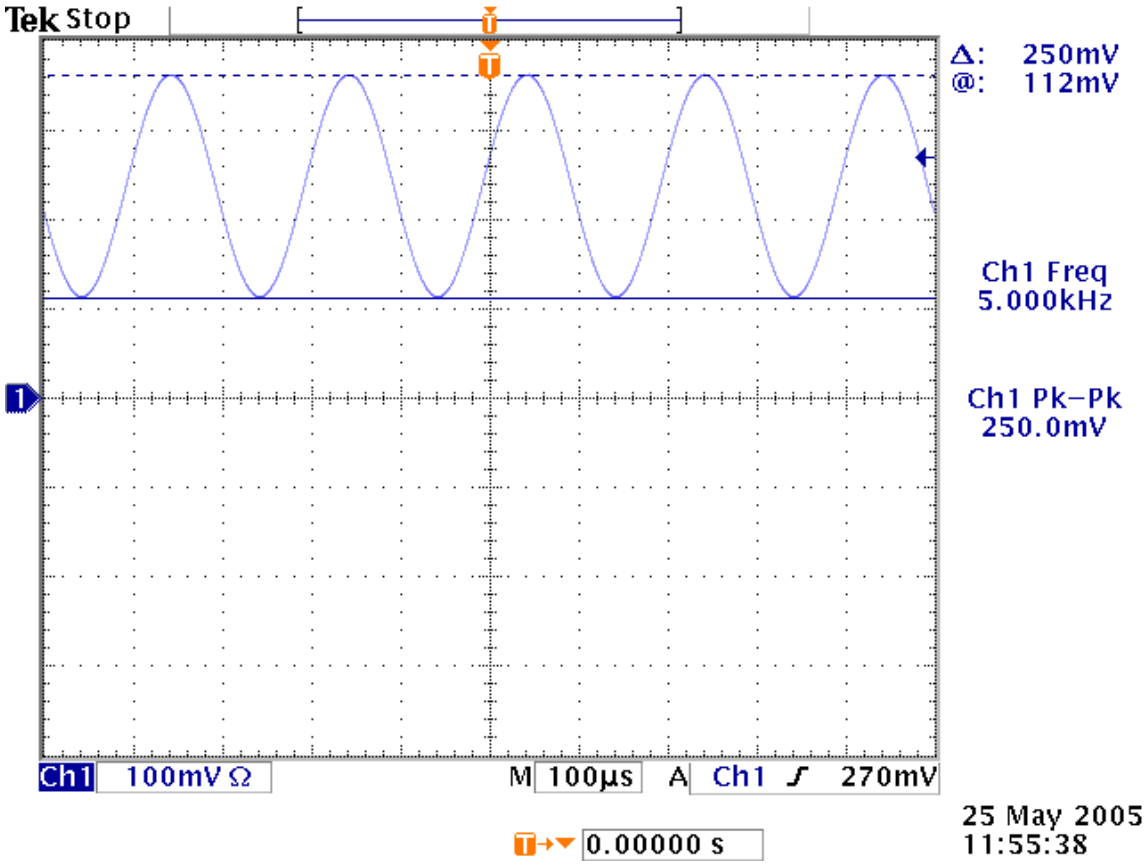
Problem:

Reproduce each of the following figures as closely as possible, and include a screenshot in your write up. Don't forget to include the measurements and cursors. Some of the input signals have introduced an offset.

If you are using the **Tektronix TDS 3012B** and it has an ethernet connection, you can easily take screenshots of the display. Simply type the IP address of the oscilloscope into your computer's web browser, and it will connect to the scope's web interface. If you don't know the IP address of your scope, restart it. The 3012B will display its current address during it's start up sequence. Once you're connected to the scope from your web browser, you may save the image of the current display or control it through the web interface.

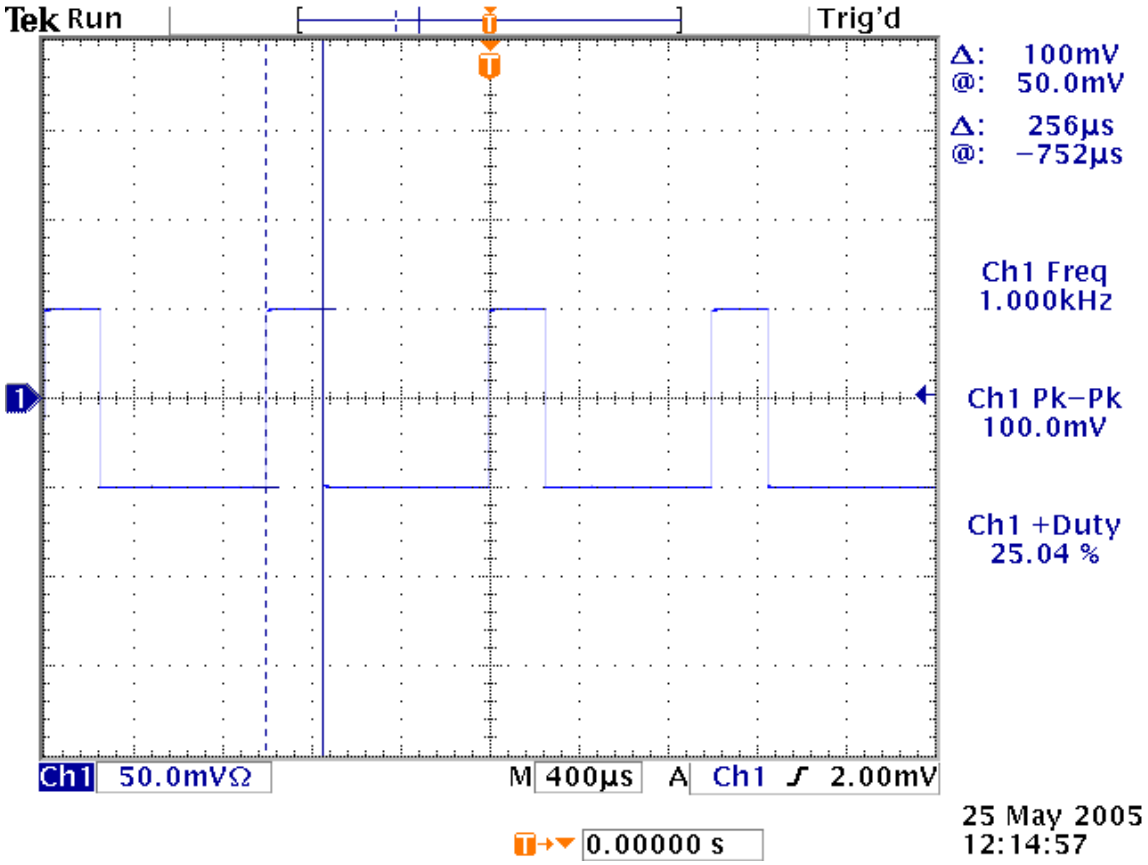
By default, the oscilloscope acquires the signal with a very rough sampling process. If you wish to improve the quality of your display you may change the Acquisition Mode. You may do this by going to **Quick Menu**-> **Acquire Mode** then changing the mode. Averaging several samples will give you a much smoother looking result.

Figure 1: Sine Wave



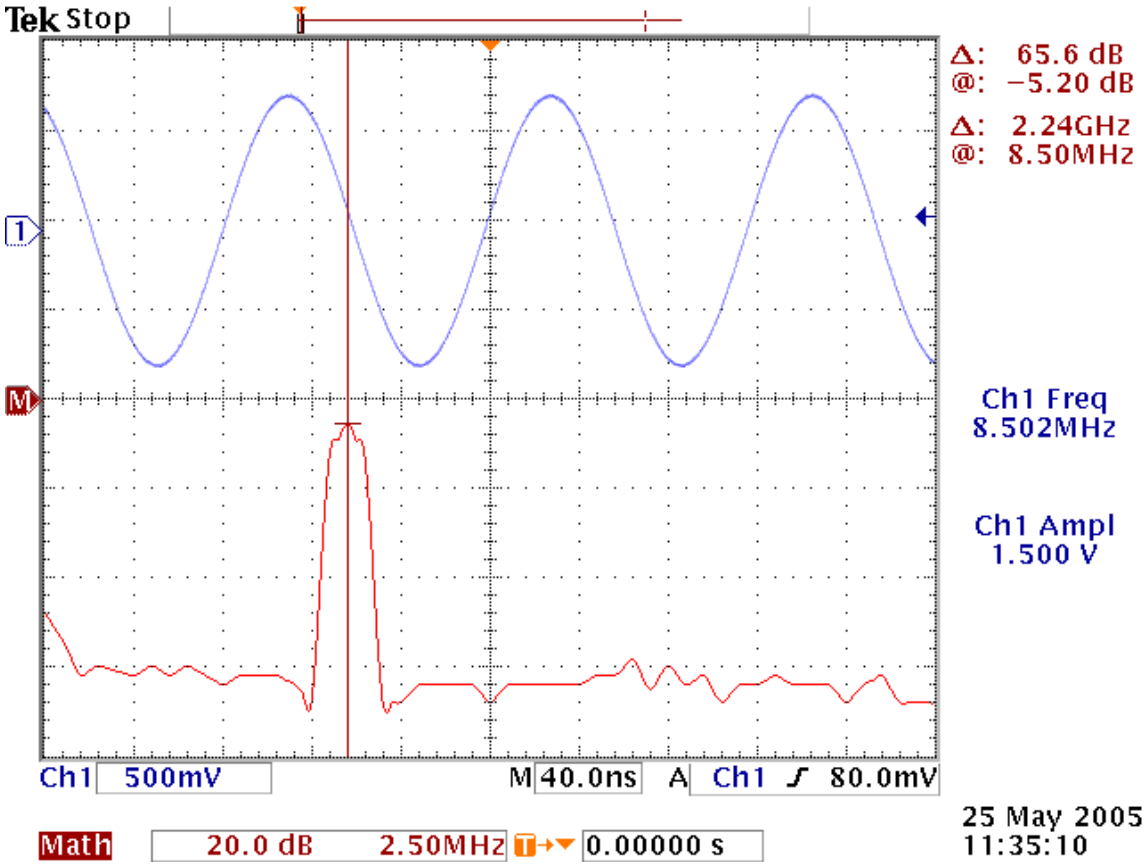
Sine wave with measurements and cursors

Figure 2: Square Wave



Square wave with measurements and cursors

Figure 3: FFT of Sine Wave



Sine wave and it's FFT. FFT options may be found under the **Math** section.

Exercise:

Problem:

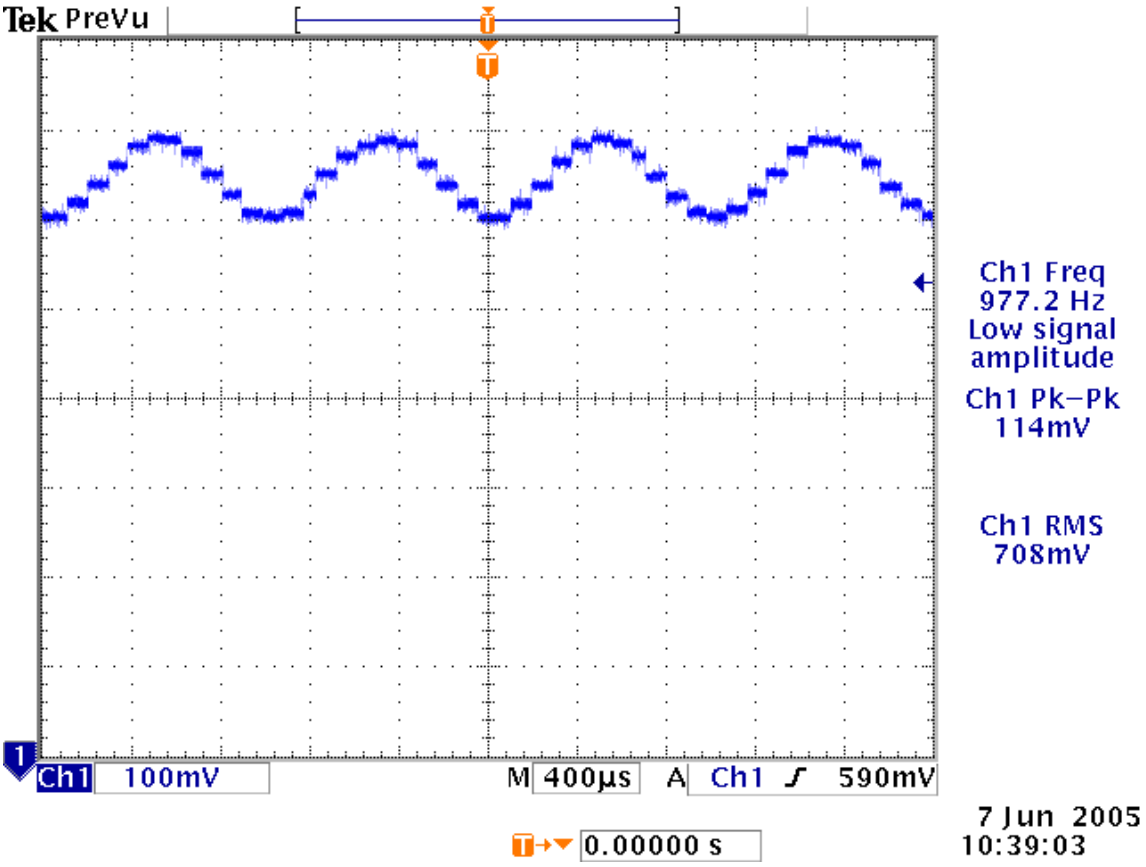
Now we will measure a couple signals directly on the ELEC226 board and learn how to read a schematic. Download lab1_2_lite.hzx. Open CrossStudio Release 1.3, and set your target as the MSP430 USB CrossConnect. You can do this by clicking on **View**-> **Targets**. The Targets window should open up on the right panel. Right click on **MSP430 USB CrossConnect** and select **Connect**. You may download the project file to the MSP by selecting **Download File**-> **Download CrossWorks Executable File** and navigating to the desired file.

Now locate the two high frequency crystal oscillators. They are small silver cylinders. Directly probe the two oscillators, and measure their respective frequencies. Next, using the schematic locate the pin that outputs **SMCLK** or **MCLK**. (Use the pin on Port 5.) This is currently configured to output one of the oscillators. Probe this pin and compare with the results that you obtained probing the oscillators directly. Next, using the schematic locate the output pins for **DAC0**. Measure the peak-to-peak amplitude, and frequency of the signal and include a screenshot.

Exercise:

Problem:

Next, we will run a couple signals from the function generator through the board's Analog to Digital Converters (ADC) and measure it a couple different places. Download lab1_3_lite.hzx. Download the file like you did in Problem 2. Set your function generator to produce a signal less than 2 kHz. Connect it to the input of **A1**, analog input 1, and measure the signal at the output of **DAC0**. Try to recreate the following figure.



Sine wave through ADC and DAC

What happens are you increase the frequency or amplitude of the input signal? At what frequency does the signal very become skewed?

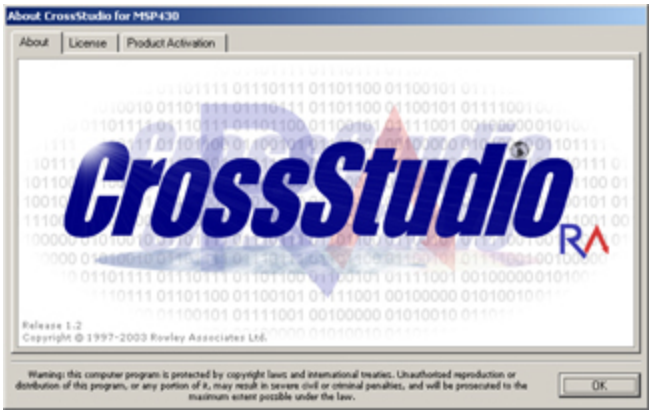
What is a program?

A **program** is a set of instructions that are grouped together to accomplish a task or tasks. The instructions, called **machine code** or **assembly code** consist of things like reading and writing memory, arithmetic operations, and comparisons. While these instructions sound simple, it is actually possible to solve a huge group problems with them. The difficulty in doing so is that you must specify in exact detail precisely how. Good programming is both an art and a science, and what you will learn today is a beginning of the craft.

As mentioned above, the individual instructions that the machine actually quite simple or **low-level** in computer parlance. Writing complex programs in assembly code took such a long time that eventually better **programming languages** were invented. A programming language, like C, is a formal set of grammar and syntax like assembly code; but the instructions in **high-level** languages encompass hundreds of assembly instructions. Programs called **compilers** translate a program written in a higher level language into assembly so that the computer can actually execute the instructions. Compilers let the programmer write programs so that humans can read them easily while the computer can still execute the instructions.

Generally programming code is organized into text files with suffixes that indicate the programming language. In the case of C these files are appended with .c, and a C program is made up of at least one of these files.

Introduction to CrossStudio MSP430 IDE
CrossStudio MSP430 IDE

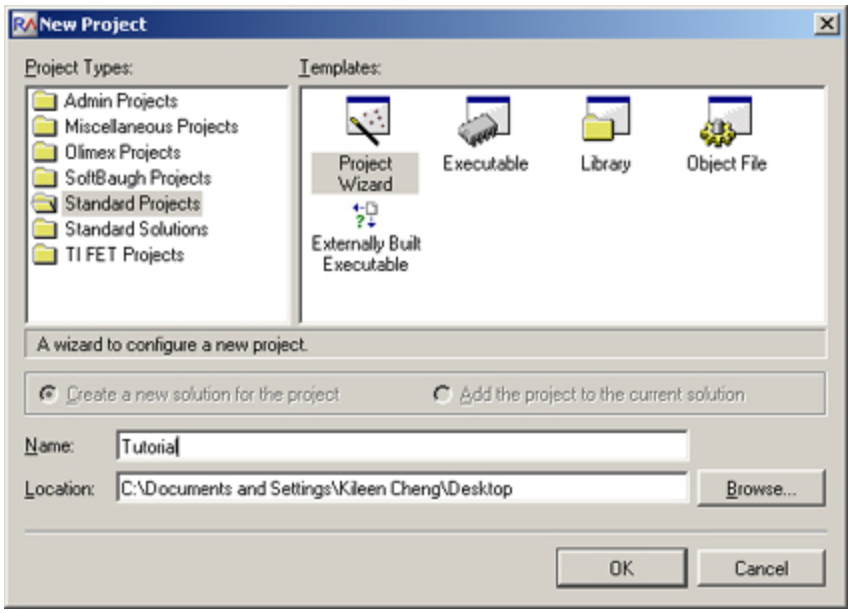


Motivations

To develop applications to run on the MSP430 chip, we use the CrossWorks MSP430 IDE (integrated development environment). Not only does this application provide a powerful code editor, but it also allows a simple one-click deployment of the source code onto the MSP chip using USB as well as hardware debugging capabilities that allow you to trace through actual stack calls. This module is intended to get your started using CrossWorks quickly so that you may begin building your own MSP430 applications!

Create a Project

The very first thing you must do before you can start downloading any code onto the MSP, is to create a project in CrossStudio that will contain all of the relevant files for your application. Select File->New->New Project and the New Project dialog will appear. By default, the Standard Projects project type will be selected in the left window pane and you will see templates such as Project Wizard, Executable, Library, etc. Select the Project Wizard template within the Standard Projects project type. Enter in a name in the text field and make sure the location field is set to the correct directory for this project.
Project Creation Wizard

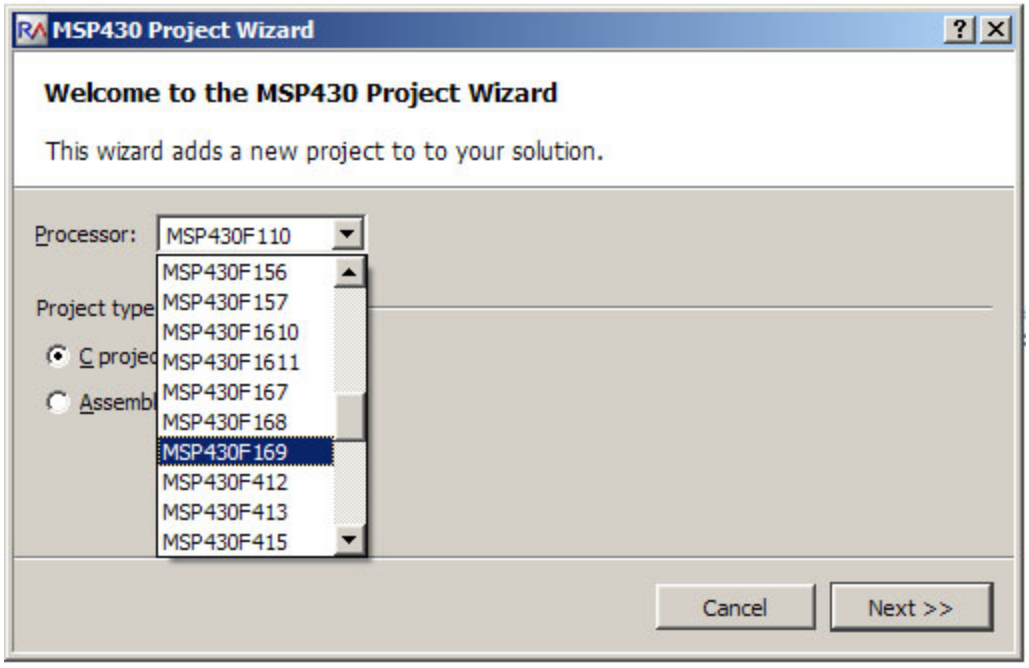


In the Standard Projects project type folder, select the Project Wizard template to create the workspace for your application.

If you have an existing solution loaded, then you can put the project into that solution or create a new solution. Don't worry if you do not know what a solution is, as it will be covered later in this tutorial. Click OK to create the project.

The project wizard will now guide you through creating your project. You will see a dialog box in which you need to customize the project based on the processor and project type.

Adding Your Project



Make sure you choose the right processor for your application! Here, I've selected the MSP430F169 since that is the chip I plan to use.

Click next and finish creating the project. The project explorer will now show the solution and the project you have just created. You'll notice that the project name is highlighted -- this is now the active project and subsequent build and debug operations will use this project. If you have more than one project then you can set the active project using the combo box on the build toolbar or the context menu of the project explorer.

Adding Files to the Project

If your project consists of more than one file, you will need to add it to the current project. To create a new file, go to Project->Add New File... If the file already exists, then choose **Project->Add Existing File**...

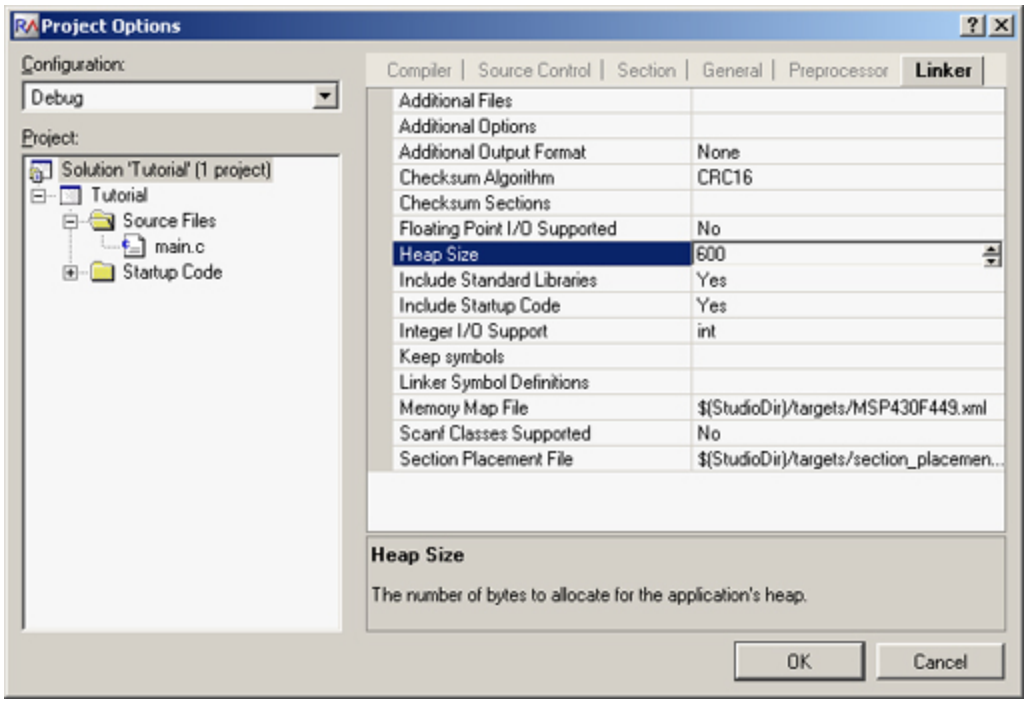
Building the Solution

The CrossStudio compiler allows you to have multiple projects in different configurations all residing in a given solution. Usually, the projects differentiate themselves as a debug build or as a release build. Debug builds will have different compiler options. Configurations can also be used to produce variants of software. For example, a library could be built for several different hardware variants. Note that configurationsn inherit properties from other configurations, enabling a single point of change for definitions that are common to configurations. A particular proplerty can be overriden in a particular configuration to enable configuration specific settings.

Upon creation of a solution, the Debug and Release configurations are generated automatically.

Setting Heap Size

The next important step that you must take is to make sure that there is heap available for the project to use. By default, there is no heap allocated. If your program will be mallocing and freeing memory dynamically, then you will need to change the heap size to a more appropriate value. To do this, choose menu item **Project**->**Properties**... You will then see a dialog box with various project options. In the left pane, click on your solution and then the linker tab on the right pane. There will be a Heap Size category that you can click on to change. The MSP430 has up to 2000 bytes of RAM that can be allocated for the heap.
Changing the Heap Size



By default, the heap size is set to zero. Here, I have increased it to 600 bytes.

The heap size is an option that is set for a given solution and thus applies to all projects that are contained within a solution. Keep this in mind if your solution contains multiple projects.

Connect to the Target

Before any of the code can be executed, you need to first connect to a target. Make sure the board is connected via USB to the computer.. Then, go to **Target->Connect MSP430 USB CrossConnect**. CrossStudio should be able to connect rather quickly to the MSP. If the application hangs, then check that the USB connection is secure, that the programming cables are plugged in correctly, and that power is being supplied to the device. If problems continue, try resetting the device, unplugging everything for awhile, and starting over again.

Run Your Program!

This is the part you've been waiting for: actually testing your program! Use the **Debug** -> **Start Debugging** menu item to load the currently active project and set your program running. You can also set any breakpoints beforehand; by default, there are no breakpoints set by the debugger.

You can pause the debugger when the target is running if you would like to look at the value of certain variables. Select **Debug** -> **Break** and open a watch window to examine the value of certain variables. It is suggested that you copy the variables you are interested in to temporary global variables. Because local variables go out of scope, it is uncertain if their correct value is maintained when the debugger is paused.

Debugging can be stopped using **Debug** -> **Stop**. At this point, the watch window will not display your variable values. At this point, I would suggest that you make any modifications to your program as necessary and restart the debugger from the beginning. It is possible to resume debugging by using **Debug** -> **Go**, but it is not recommended.

Good luck!

You're all set to start using the CrossStudio compiler to write embedded microcontroller code.

Configuring Digital I/O

Digital I/O such as the LEDs and pushbuttons are configured by modifying a several registers pertaining to the port that they are attached to. Check the datasheet to find the respective port number of the peripheral you which to control. For more detailed information about Digital I/O on the MSP430 check **Chapter 9: Digital I/O** of the User's Guide.

First, we must assign the direction of the corresponding I/O pins. This is done by setting the direction register, **PxDIR**, with the appropriate bit. By default, all I/O pins are assigned to be inputs.

- Bit = 0: The port pin is switched to input direction.
- Bit = 1: The port pin is switched to output direction.

On the MSP430F16x Lite Development Board the three LEDs are located on Ports 1 and 2. The port number will correspond the to x value in registers such as **PxIN**, **PxOUT**, or **PxDIR**. Therefore, if we wanted to define the direction of a pin on port 2 we would write to **P2DIR**.

Exercise:

> **Problem:**
>
> How do we switch the three pins (P1.7, P2.2, and P2.3) corresponding to the LEDs to be outputs?

> **Solution:**
>
> `P1DIR |= 0x80; P2DIR |= 0x0C;` 0x0C in hex is equivalent to 0b00001100 in binary. Similarly, 0x80 = 0b10000000. You may refer to the module about Binary and Hexadecimal Notation to learn how to do this conversion or you could use the Windows Calculator (**Start** -> **Run...** -> **Calc**) to do it much more quickly. We can now easily see that P1.7, P2.2, and P2.3 are set to 1 which makes them outputs.

Note: It is helpful to use `|=` instead of `=` because it won't overwrite bits that are not set to 1 in the mask.

Output pins may be toggled using the **PxOUT** register. LEDs are turned on by setting their corresponding register bits low.

Exercise:

Problem:

How would be turn on the three LEDs without modifying any other bits in the register?

Solution:

`P1DIR &= ~0x80; P2DIR &= ~0x0C;` This will turn on all three LEDs assuming they had already been set to be outputs.

Note:~0x0C = 0xF3 = 0b11110011

Since all I/O registers are set as inputs by default we do not have to set the direction of the push buttons. Each time an input is toggled a bit in **PxIN** will be modified.

Exercise:

Problem:

Write a couple different polling schemes for detecting if BUTTON_1 was pushed.

Note: PxIN bits corresponding to the push buttons are high by default (i.e. the button is not depressed.)

Solution:

`while(!(P2IN&0x01));` or `if (!(P2IN&0x01));`

Exercise:

Problem:

Now we will write a program that lights up one of the LEDs and will light up a different LED once BUTTON_2 is pressed. The LED sequence should go as follows: red, green, yellow, repeat.

Create a new project in CrossStudio and make sure you select the correct processor, the MSP430F169.

Include the correct header file by adding the following line at the top of the main.c file. `#include <msp430x16x.h>`

It may be helpful to define some macros for commonly used register values. For example, if we add `#define red_on ~0x04` to the top of the file (after the `#include`) we may call `red_on` every time we wanted the value ~0x04. Similarly, you may write a function to turn a light on or off.

Complete the program.

Solution:

```
#include <msp430x16x.h> void main(void) { P1DIR |= 0x80; // Set P1.7 as an output pin P2DIR |= 0x0C; // Set P2.2 & P2.3 as output pins P1OUT |= 0x80; P2OUT |= 0x0C; // Turn off all LEDs while(1){ P1OUT |= 0x80; // turn off yellow LED
```

```
P2OUT &= ~0x04; // turn on red LED
while(P2IN&0x02); // waits here while button
isn't depressed while(!(P2IN&0x02)); // waits
here while button is pushed in P2OUT |= 0x04;
// turn off red LED P2OUT &= ~0x08; // turn on
green LED while(P2IN&0x02); while(!
(P2IN&0x02)); P2OUT |= 0x08; // turn off green
LED P1OUT &= ~0x80; // turn on yellow LED
while((P2IN&0x02)); while(!(P2IN&0x02)); } }
```

Setting Breakpoints in Crossworks

C with an embedded controller does not have as many input-output (IO) features as a regular computer. To help you debug, it will sometimes be necessary to stop the processor while it is running and examine the state of the system. To accomplish this we will use **breakpoints**. A breakpoint is a specific command to the development environment to stop execution of the processor when a certain condition happens. These conditions range from when a certain instruction is reached to when certain data is written or read. The advanced options are broad.

To set a basic breakpoint, one which will stop execution when a certain line is reached, just click on the left margin of the C file on the line you want to trigger. A red dot should appear to indicate you have set a break point. Click once more to make it go away. Crossworks keeps track of all of the breakpoints for you. To see this information go to Debug->Debug Windows->Breakpoints this will pop up a list of all of the breakpoints you currently have enabled. The window also has buttons to create, delete, and modify breakpoints. There are several different types of breakpoints and each one is configured differently. We will use the same terminology for the breakpoints as the Rowley environment does, but the usage is not standardized.

A source code breakpoint triggers on arrival at a certain instruction in the source code. This kind of breakpoint can be created simply by left clicking on the left margin of the line in question. The breakpoint should then appear in the breakpoint window. Right clicking on the entry for a breakpoint allows you to edit it. For all breakpoints, you may select that they only trigger on a certain iteration by editing the Counter field. If left blank, the breakpoint will trigger each time it occurs. Entering a number into the counter field will trigger the breakpoint on that numbered time the event occurs.

Another kind of breakpoint is an expression breakpoint; it triggers when a certain function is executed or a variable is written to. To set a basic expression breakpoint highlight the name of the function or variable in question and right click on the item to bring up a menu. From this menu

select “Set Breakpoint On <name of variable or function>.” You can still edit the breakpoint to use the counter.

Ranged breakpoints watch for data accesses and execution of instructions inside or outside of a specified memory range.

Finally valued breakpoints break when certain data is written to a variable. The mask capability lets you only look at a certain bit set of the value, and the comparison values allow you to select from a range of values to stop on. To set up a valued breakpoint, create an expression breakpoint for a variable and edit the breakpoint to be valued. Make sure that you select whether you anticipate the conditions being a write or a read.

Alternate explanations of breakpoints can be found in the help contents of the Crossworks system.

Lab 2: C and Macros with Texas Instruments' MSP430

With the MSP430, With the MSP430, the primary difference between "normal" C and programming C in the embedded space is that you will need to write to registers directly to control the operation of the processor. Fortunately, the groundwork has already been laid for you to make this easier. All of the registers in the MSP430 have been mapped to macros by Texas Instruments. Additionally, the important bit combinations for each of these registers have macros that use the same naming convention as the user's guide. Other differences from the C used on most platforms include:

- Most registers in the MSP are 16 bits long, so an `int` value is 2 bytes (16 bits) long.
- Writing to registers is not always like writing to a variable because the register may change without your specific orders. It is always important to read the register description to see what the register does.
- The watchdog timer will automatically reset the MSP unless you set the register not to.
- There is only a limited "standard out" system. Standard out will typically print results to your computer screen. The board you have been provided does have 7 segment displays and LED's but this does not allow for a full debugging display that a PC would have.
- Floating-point operations cannot be efficiently performed. In general, you should avoid floating point decimal calculations on the MSP because it does not have special hardware to support the complicated algorithms used.

Exercise:

Problem:
Code Review

In this exercise, you may want to use some of the debugging tools to help you understand what the code is doing. You may use any of the following items:

- **Breakpoints -** Left click on the arrow next to the line of code you want to break on. The program will stop when it gets to this point

and you are able to view the contents of memory and current variable contents.

- **Watch Window** - Right click on the variable you want to monitor and select **Add "variable" to Watch**. Variables that have been modified since the last time that the program was stopped will turn red.
- **Locals Window** - From the **View** menu, click **Other Windows** then **Locals**. The Locals Window will automatically display the values for all local variables that are currently being used. Variables are considered "local" if they are within the function that is currently being processed.

Start a new project. Cut and paste the following code into main.c:

```
#include <msp430x16x.h> #include
<__cross_studio_io.h> void main(void){ int
i,j,tmp; int a[20]=
{0x000C,0x0C62,0x0180,0x0D4A,0x00F0,0x0CCF,0x0C
35,0x096E,0x02E4,
0x0BDB,0x0788,0x0AD7,0x0AC9,0x0D06,0x00EB,0x05C
C,0x0AE3,0x05B7,0x001D,0x0000}; for (i=0; i<19;
i++){ for (j=0; j<9-i; j++){ if (a[j+1] < a[j])
{ tmp = a[j]; a[j] = a[j+1]; a[j+1] = tmp; } }
} while(1); }
```

1. Explain what this program is doing.
2. Use any of the methods listed above to show the updated array. What is the final result?
3. Modify the code so that it prints the final version of the array to standard out (the display window). What are the drawbacks and benefits of using this over setting a breakpoint?

Note: To use the standard out, add the following line to the top of your code: `#include <__cross_studio_io.h>` The `debug_printf()` function will print to standard out. For example, `debug_printf("x equals %d\n", x);` will print out the

value of x to the window. The `%d` means that x is a number, and `\n` will produce a line break.

Exercise:

Problem:
Functions

Multiplications and division are very complex operations to do on any microprocessor. The operations should be avoided if possible or should be replaced with simpler, equivalent operations.

1. What do the operators << and >> do?
2. How could you use these operators to perform multiplication and division?
3. Write the function `multiply(int x, int y)` that takes parameter `x` and multiplies it by `y` by using a bit shift. It must return an `int`. For symplicity, it is OK to assume that `y` is a power of 2.
4. Next, write the function `divide(int x, int y)` that takes parameter `x` and divides it by `y` by using a bit shift. It must also return an `int`.

Exercise:

Problem:
Digital I/O Registers

Open the file **msp430x16x.h** which should be located in `C:\Program Files\Rowley Associates Limited\CrossWorks MSP430 x.x.x\include`. This file contains the macros and register definitions for the MSP430F169 we use in this class. Using the MSP430 User's Guide and the msp430x16x.h file, please answer the following questions.

1. The Watchdog Timer will automatically reset the hardware if it isn't periodically reset or disabled entirely. Usually, we will simply disable it. It can be disabled by writing to the `WDTPW` (watchdog timer password) and `WDTHOLD` (watchdog timer hold) section of the Watchdog Timer Control Register (`WDTCTL`). Refer to **Section 10.3** of the User's Guide for more information. Find the macros for this register in the msp430x16x.h file. How are they different from their description in the User's Guide? Finally, write the C code required to disable it.
2. What are the differences among P1DIR, P1SEL, P1OUT, P1IN?
3. Some port pins have multiple functions to output and it is up to the user to select the appropriate signal. Write some code that would select the alternate function of P2.2 (pin 2 of port 2). What will the result be on our hardware?

Exercise:

Problem:
Programming Digital I/O

Write a program to do the following:

1. When the program starts all three LED's should light up for about a second then turn off.
2. Next, the green LED should blink for about 1/2 second on, 1/2 second off while the other two LED's are off. Use long for-loops to generate a delay.
3. Pushing Button 1 should cause the green LED to stop blinking and cause the red LED to start the blinking pattern.
4. Pushing the button again should continue the pattern with the yellow LED, and pushing it more times should repeat the green, red, yellow pattern.
5. The current LED that's blinking should stop as soon as the button is pressed and the next LED should begin immediately.

Note: Make sure you disable the watchdog timer at the very beginning of the program. Refer to Introduction to programming the MSP430 to learn how to enable and use the pushbuttons and LEDs.

You will need to demonstrate this code to the labbie, turn in a hard copy of your program, and post it online.

Introduction to Assembly Language

Assembly Language

Assembly language, commonly referred to as assembly, is a more human readable form of machine language. Every computer architecture uses its own assembly language. So processors using an architecture based on the x86, PowerPC, or TI DSP will each use their own language. **Machine language** is the pattern of bits encoding a processor's operations. Assembly will replace those raw bits with a more readable symbols call **mnemonics**.

For example, the following code is a single operation in machine language. `0001110010000110` For practical reasons, a programmer would rather use the equivalent assembly representation for the previous operation. `ADD R6,R2,R6 ; Add $R2 to $R6` This is a typical line of assembly. The **op code** `ADD` instructs the processor to add the **operands** `R2` and `R6`, which are the contents of register R2 to register R6, and store the results in register R6. The "`;`" indicates that everything after that point is a comment, and is not used by the system.

Assembly has a one-to-one mapping to machine language. Therefore, each line of assembly corresponds to an operation that can be completed by the processor. This is not the case with high-level languages. The **assembler** is responsible for the translation from assembly to machine language. The reverse operation is completed by the **dissasembler**.

Assembly instructions are very simple, unlike high-level languages. Often they only accomplish a single operation. Functions that are more complex must be built up out of smaller ones.

The following are common types of instructions:

- Moves:
    - Set a register to a fixed constant value
    - Move data from a memory location to a register (a load) or move data from a register to a memory location (a store). All data must

be fetched from memory before a computation may be performed. Similarly, results must be stored in memory after results have been calculated.
    - Read and write data from hardware devices and peripherals

- Computation:

    - Add, subtract, multiply, or divide. Typically, the values of two registers are used as parameters and results are placed in a register
    - Perform bitwise operations, taking the conjunction/disjunction (and/or) of corresponding bits in a pair of registers, or the negation (not) of each bit in a register
    - Compare two values in registers (>, < , >=, or <=)

- Control Flow:

    - Jump to another location in the program and execute instructions there
    - Jump (branch) to another location if a certain condition holds
    - Jump to another location, but save the location of the next instruction as a point to return to (a call)

Advantages of Assembly

The greatest advantage of assembly programming is raw speed. A diligent programmer should be able to optimize a piece of code to the minimum number of operations required. Less waste will be produced by extraneous instructions. However, in most cases, it takes an in-depth knowledge of the processor's instruction set in order to produce better code than the compiler writer does. Compilers are written in order to optimized your code as much as possible, and in general, it is hard to write more efficient code than it.

Low-level programming is simply easier to do with assembly. Some system-dependent tasks performed by operating systems simply cannot be expressed in high-level languages. Assembly is often used in writing **device drivers**, the low level code that is responsible for the interaction between the operating system and the hardware.

Processors in the embedded space, such as TI's MSP430, have the potential for the greatest gain in using assembly. These systems have very limited computational resources and assembly allows the maximum functionality from these processors. However, as technology is advancing, even the lowest power microcontroller is able to become more powerful for the same low cost.

Structure of an Assembly Program

The assembly program begins execution at the reset interrupt. The reset interrupt is the first thing that occurs when power is given to the processor. By default in the Rowley files, the reset interrupt is loaded to send the execution of the program to the start of the .code segment. Until a branch is reached, the processor will execute each instruction in turn. If the program does not loop back to an earlier point to keep going, eventually the execution will reach the end of the valid instructions in memory. You should never let this happen.

The control of a programs execution is called **control flow**, and it is accomplished through branching, jumping, function calls, and interrupts. Interrupts are the subject of future labs. Branching and jumping refer to changing the next instruction from the next one sequentially to an instruction elsewhere in the program. By branching to an instruction above the branch itself you can cause the program to repeat itself. This is a basic loop in assembly. Branches can also be conditional. In the MSP architecture conditional branches are generally dependent on the status register (SR) bits to decide whether to execute the next instruction after the branch or the instruction the branch specifies. Many arithmetic and logical operations can set the relevant bits in the status register; check the MSP430 User's Guide for which ones you will need.

Once you understand the basics of assembly you should be able to write some simple routines.

Lab 3: Introduction to Assembly Language

In this lab you will be introduced to assembly programming. The specific assembly language instruction set of the MSP will be explained. Setting up and executing assembly projects in Crossworks will also be explained. Finally you will implement the blinking lights program from the previous lab in assembly.

Read the following modules before you begin lab exercises:

- CPU Registers in the MSP430
- Structure of an Assembly Program

Exercise:

Problem: Formulate instructions to do the following things:

1. Set bit 3 to 1 at the memory address 0xd640 while leave bits 0-2 and 4-16 unaffected.
2. Jump to the instruction labeled POINT if the carry bit is set.
3. Shift register R6 right one place while preserving the sign.

Exercise:

Problem:

Examine this loop: `... more instructions... Mov.w &I, R4 Cmp.w #0x0000, R4 JZ After_loop Start_loop: Dec.w #0x0001, R4 JZ After_loop BR #Start_loop After_loop: ...more instructions...`

- How many times will this loop execute?
- Why do we use the BR instruction with a #Start_loop, but with the JZ we use a plain After_loop?
- What does the first JZ instruction do? If we did not have this initial Cmp and JZ, what (possibly) inadvertent effect might occur?

Exercise:

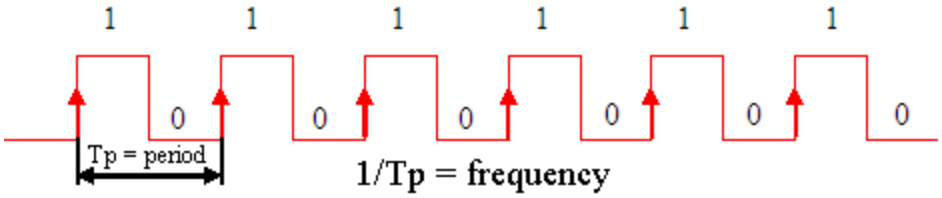
Problem:

Re-write the blinking light program from Lab 2, now using assembly code. The program should do the following:

1. When the program starts all three LED’s should light up for about a second then turn off.
2. Next, the green LED should blink for about 1/2 second on, 1/2 second off while the other two LED’s are off.
3. Pushing Button 1 should cause the green LED to stop blinking and cause the red LED to start the blinking pattern.
4. Pushing the button again should continue the pattern with the yellow LED, and pushing it more times should repeat the green, red, yellow pattern. Note that the current LED that's blinking should stop as soon as the button is pressed and the next LED should begin immediately.

What is a digital clock?

The clock of a digital system is a periodic signal, usually a square wave, used to trigger memory latches simultaneously throughout the system. While no part of this definition is strictly true, it does convey the basic idea. Square waves are used because the quick transitions between high and low voltages minimize the time spent at uncertain digital levels. The clock ideally reaches all parts of the system at the same time in order to prevent sections from getting out of sync. Clock signals are generally periodic because the user wants to run the system as fast as possible, but this is often not a necessary attribute.



Clock signals are used to synchronize digital transmitters and receivers during data transfer. For example, a transmitter can use each rising edge of the clock signal of Figure 1 to send a chunk of data.

A faster clock rate all means that you can process more instructions in a given amount of time.

Clock System on the MSP430

The clock system on the MSP430 is designed to be flexible and low power. The implementation of these goals is largely based on the ability to select different clocks for different parts of the chip. By choosing the minimum clock speed necessary for a given module, power consumption is reduced and the particular synchronization needs of the module can be met.

The MSP has three clock sources for the clocking system, and three clock lines that chip systems can choose between. The clock sources are used as the basis for the clock lines, and this allows for a mix of slow and fast clocks to be used around the system. The three clock sources are:

1. **Low Frequency Crystal Clock (LFXTCLK)** – this signal is meant to come from a watch crystal external to the MSP itself. The crystal connects to the XIN and XOUT pins, and the intended oscillation is 32kHz. This is the source of the Auxiliary Clock line (ACLK). The primary control of this source is that it can be turned off with the OSCOFF option in the Status Register. The source also has a high-speed mode for faster crystals.
2. **Crystal 2 Clock (XT2CLK)** – this signal is the second external clock source, and it is connected to the XT2IN and XT2OUT pins. In our case, the source is connected to a 7.3MHz crystal. In general, this signal is meant to be the high-speed clock source. This source can be turned off with the XT2OFF bit of the Basic Clock system control register 1 (BCSCTL1).
3. **Digitally Controlled Oscillator Clock (DCOCLK)** – this is the only internally generated clock input, and it is the default clock source for the master clock upon reset. By default this clock runs at about 900kHZ, but the RSELx, MODx, and DCOx bits allow this to be divided down or even blended to achieve a lower clock frequency on average.

The three clock lines are:

1. **Master Clock (MCLK)** – This clock is the source for the MSP CPU core; this clock must be working properly for the processor to execute instructions. This clock has the most selection for its source. The

source is selected with the SELMx bits of the Basic Clock System Control Register 2 (BCSCTL2). The divider is controlled with the DIVMx of the BCSCTL2. Finally, the CPU can be turned off with the CPUOFF bit of the Status Register (SR), but to recover from this state an interrupt must occur.

2. **Submaster Clock (SMCLK) -** This clock is the source for most peripherals, and its source can either be the DCO or Crystal 2. The source clock is controlled with the SELS and SCG bits of the BCSCTL2 and SR. The divider is controlled by the DIVSx bits of the BCSCTL2.
3. **Auxiliary Clock (ACLK) -** this clock line's source is always LFXTCLK. It is an option for slower subsystems to use in order to conserve power. This clock can be divided as controlled by the DIVAx bits of the Basic Clock System Control Register 1 (BCSCTL1).

The MSP clock system has dividers at the beginning of its clocks, and at most peripheral destinations. This allows for each module to keep a separate timing scheme from other modules. This is often necessary for off chip buses because these systems have to meet a variety of speed requirements from the outside. For educational purposes the fastest clocks are usually the most useful, but power consumption is the primary cost of high speed clocks.

Clock Dividers

Throughout the MSP clocking system there are clock dividers. A clock divider reduces the frequency of an input clock and outputs this divided frequency. The simplest dividers work on multiples of two, so the output might be a square wave of half, one quarter, or one eighth the input square wave's frequency.

Lab 4: Clocking on MSP430

Exercise:

Problem:

Clock Setup

The following exercise will show you how to manipulate the clocking system on the MSP430. You may need to refer to the MSP430F16x Lite Development Board schematic and the User's Guide to correctly configure the clock as specified.

- In order to easily check the state of the clock, output MCLK from a pin header. (HINT: Output MCLK from P5.4) Use the oscilloscope to observe the frequency of the clock, and to see the impact of the changes you will make. Without modifying the clock registers any further, at what clock rate is the processor running at? How is the clock currently configured in order to produce this built-in clock signal?
- Take the following piece of code that attempts to configure the clock on the MSP430. It has one major caveat: it doens't work. It contains incorrect Macro definitions, incorrectly sets the appropriate bits one some registers, and most importantly, it doesn't enable that fastest oscillator for SMCLK and MCLK. Modify the appropriate lines in order to operate 7.37 MHz without deleting any lines of code.

```
BCSCTL1 |= XTOFF; // XT2 = HF XTAL do { IFG1 = OFIFG; // Clear OSCFault flag for (i = 0xFF; i > 0; i--); // Time for flag to set } while ((IFG1 & OFIFG) != 0); // OSCFault flag still set? BCSCTL2 &= SELM0+DIVS1; // Enable fastest clocking mode
```

The preceding code should make the following happen:

- XT2 must be turned on and is set as the high frequency oscillator.
- Clear the oscillator fault interrupt flag. It is helpful to halt the processor until the processor is certain that the flag is cleared and

the crystal has stablized. This is done by setting the flag bits in the Interrupt Flag Register to `1`.
- Don't forget to stop the Watchdog Timer.

The power consumed by any computer system is directly proportional to it's clock speed. A new Pentium 4 can consume upwards of 100 Watts of power. Embedded devices try to consume as little power as possible, so it will only enable the features it absolutely needs and will run at the lowest possible clock frequency.

- Change the configuration of the clock so that it uses the DCO and runs at ~200kHz. Why wouldn't we want to run at this rate all the time?

Exercise:

Problem:
LED update

Using your new (fast) timer settings and your blinking light system from Lab 2 Problem 4, modify the code so that each LED remains on for a half second. Basically, generate the same behavior as the original version but using the new clock settings. What were the original (much slower) settings for the clock? What was changed in order to keep the original blink rate?

Exercise:

Problem:
Calculator

Write a program that does that following:

- Write a funtion `void led_num(int num)`. The function will take a number as a perameter and will use the LEDs to display this number in binary. Be sure to check for overflow. Overflow occurs when you try to represent a number greater than the number of bits available. In the case of the LEDs, you can not represent a number greater than 3 bits. What is that number? You

should put all the possible values for PxOUT in an array, in order to simplify your code.

The main program will do the following:

- Each time button_1 is pushed the LEDs will count up (in increments of one) from zero to their maximum.
- Each time button_2 is pushed the number currently displayed on the LEDs is added to an internal accumulator and the results will be displayed.
- If button_1 is pushed again, the LED counter will count up from zero, and that number will be added to the running sum when button_2 is pushed.
- Pushing the reset button will let you start over.

Interrupts

An **interrupt** is an event in hardware that triggers the processor to jump from its current program counter to a specific point in the code. Interrupts are designed to be special events whose occurrence cannot be predicted precisely (or at all). The MSP has many different kinds of events that can trigger interrupts, and for each one the processor will send the execution to a unique, specific point in memory. Each interrupt is assigned a word long segment at the upper end of memory. This is enough memory for a jump to the location in memory where the interrupt will actually be handled. Interrupts in general can be divided into two kinds- maskable and non-maskable. A **maskable** interrupt is an interrupt whose trigger event is not always important, so the programmer can decide that the event should not cause the program to jump. A **non-maskable** interrupt (like the reset button) is so important that it should never be ignored. The processor will always jump to this interrupt when it happens. Often, maskable interrupts are turned off by default to simplify the default behavior of the device. Special control registers allow non-maskable and specific non-maskable interrupts to be turned on. Interrupts generally have a "priority;" when two interrupts happen at the same time, the higher priority interrupt will take precedence over the lower priority one. Thus if a peripheral timer goes off at the same time as the reset button is pushed, the processor will ignore the peripheral timer because the reset is more important (higher priority).

The function that is called or the particular assembly code that is executed when the interrupt happens is called the Interrupt Service Routine (ISR). Other terms of note are: An interrupt flag (IFG) this is the bit that is set that triggers the interrupt, leaving the interrupt resets this flag to the normal state. An interrupt enable (IE) is the control bit that tells the processor that a particular maskable interrupt should or should not be ignored. There is usually one such bit per interrupt, and they are often found together in a register with other interrupt enable bits. The most important interrupt on MSP430 is the reset interrupt. When the processor detects a reset or powers up for the first time, it jumps to the beginning of memory and executes the instructions there. The highest priority interrupt vector begins at the address 0xfffe. The lowest priority interrupt begins at 0xFFE0. The complete set of interrupts is ranked by priority:

- 15 non-maskable: External reset, power up, watchdog timer reset, invalid flash memory activation
- 14 non-maskable: oscillator fault, flash memory access violation, NMI
- 13 maskable: timer B capture compare register 0
- 12 maskable: timer B capture compare registers 1-6, timer B interrupt
- 11 maskable: comparator A interrupt
- 10 maskable: watchdog timer interrupt
- 9 maskable: USART0 receive interrupt, I2C interrupt
- 8 maskable: USART0 transmit interrupt
- 7 maskable: A/D converter interrupt
- 6 maskable: timer A capture compare register 0 interrupt
- 5 maskable: timer A capture compare registers 1-2 interrupt
- 4 maskable: port 1 interrupts
- 3 maskable: USART1 receive interrupt
- 2 maskable: USART1 transmit interrupt
- 1 maskable: port 2 interrupts
- 0 maskable: D/A converter interrupt

When the interrupt first occurs on the MSP there is a precise order of events that will occur. This process takes 6 instruction cycles to occur.

1. The current instruction completes.
2. The program counter as it is after the above instruction is pushed onto the stack. The stack is memory whose contents are kept in last in first out order. The stack pointer is always updated to point to the most recent element added to the stack. This allows the processor to call functions and track interrupts. When something is pushed onto the stack, the stack pointer is incremented and the pushed data is written to that location. When you copy out of the stack and decrement the stack pointer, this is called popping something off the stack.
3. The status register is pushed onto the stack.
4. The highest priority interrupt waiting to occur is selected.
5. Single source interrupts have their interrupt request flags reset automatically. Multiple source interrupt flags do not do this so that the interrupt service routine can determine what the precise cause was.
6. The status register with the exception of the SCG0 bit is cleared. This will bring the processor out of any low-power modes. This also

disables interrupts (the GIE bit) during the interrupt.
7. The content of the interrupt vector is loaded into the program counter. Specifically the processor executes the instruction at the particular memory location (the interrupt vector) for that interrupt. This should always be a jump to the interrupt service routine.

The interrupt service routine is the code that the programmer writes to take care of the work that needs to be done when a particular interrupt happens. This can be anything you need it to be. Because entering the interrupt turned off the GIE bit, you will not receive any interrupts that happen while you are still in the interrupt service routine. You can turn the interrupts back on if you need to receive interrupts during your interrupt, but usually it is a better idea to make interrupt service routines shorter instead. In C interrupts are simply functions with special declarations. You never call these functions; the compiler simply sets up the interrupt vector table to call your function when the particular interrupt occurs.

This example interrupt is pulled from the fet140_wdt01.c example file by Mark Buccini. The complete file is in the Rowley directory under samples/msp430p140_C. `// Watchdog Timer interrupt service routine void watchdog_timer(void) __interrupt[WDT_VECTOR] { P1OUT ^= 0x01; // Toggle P1.0 using exclusive-OR }`

Interrupt functions should always be void and accept no arguments. This particular interrupt service routine (ISR) is called watchdog_timer, but the name does not matter. The way the compiler knows that this function should handle the watchdog timer interrupt is what follows the function name. The __interrupt[] indicates that this is an interrupt and WDT_TIMER is a macro from the MSP header file. Every interrupt vector in the processor has a macro defined for it. To attach this interrupt service routine to a different interrupt, all you need to do is change the WDT_TIMER to one of the other macros defined in the header msp430x16x.h.

When the end of the ISR is reached the MSP executed a precise set of steps to pick up the execution of the program where it left off before the interrupt occurred. This process takes 5 cycles.

1. The status register and all previous settings pops from the stack. Any alterations to the status register made during the interrupt are wiped away.
2. The program counter pops from the stack and execution continues from where it left off.

Interrupt Enable Registers

Using interrupts successfully is not as simple as just writing an interrupt service routine and waiting for the event to occur. Because sometimes you do not want to activate every interrupt in the processor at once, the MSP allows you to mask out certain interrupts. When the triggering event first occurs, the processor checks whether the interrupt is enabled before jumping to the interrupt service routine. For most interrupts, the MSP checks the general interrupt enable bit in the status register and the particular interrupt's enable in the interrupt enable registers. If both of these have been configured to allow the interrupt, then the interrupt flag is set and the processor enters the interrupt service routine.

By default most interrupts are turned off upon reset, to use most peripheral modules you will need to set the enable bits in the interrupt enable registers and turn on the general interrupt enable. Enabling sometimes causes the interrupt flag to be set, so you should consult the User's guide on the best order to handle the enabling. Usually to properly configure the interrupt, you will also need to have set up the peripheral module in question before enabling the interrupt.

There are three categories of interrupts for the purpose of masking in the MSP430. Reset interrupts, non-maskable non-reset interrupts, and maskable interrupts.

Maskable interrupts are the lowest priority interrupts and can be turned off individually with the various interrupt enable registers or turned off as a group by setting the general interrupt enable bit (GIE) in the status register (SR).

Non-maskable interrupts are not subject to the general interrupt enable (GIE). However each non-maskable interrupt source can be controlled by a bit to specifically turn it on or off. These are the flash access violation interrupt enable (ADDVIE), external NMI interrupt enable (NMIIE), and the oscillator fault interrupt enable (OFIE). All three of these bits are located in the interrupt enable register 1 (IE1).

Reset interrupts have the highest priority and will always reset the execution of the device. The external reset can be configured to trigger either the reset interrupt or an NMI interrupt.

The interrupt enable registers (IE1 and IE2) are used to individually enable the interrupts. Refer to the MSP User's Guide and Data sheet on the specifics of each peripheral. The example code that accompanies the Rowley system and the Texas Instruments website are also very good sources of example code.

For example, the serial port USART receive interrupt is configured in the example file fet140_uart01_09600.c from Texas Instruments. The serial port interrupts are typical of the maskable peripherals. The procedures followed are drawn from the instructions and notes in the documentation. Often the relevant information may not be in one chapter or section of the guides. This is part of the reason working examples are essential to developing a working knowledge of the processor.

The fet140_uart01_09600.c file begins by turning off the reset activated by the watchdog timer. Then selects the external clock for the ACLK's source. The example writer assumes a 3.58 MHz clock, but in our case this is a 7.37 MHz clock. This would means that our actual baud rate for this example will be about 19200 baud rather than 9600. The example then executes a do-while loop that waits for the oscillator fault flags to stop being asserted. When an oscillator or crystal first receives power, it often has some instability in its oscillation. The oscillator fault flag detects the instabilities. Because we are planning on using an external clock in the example, it is prudent to wait for the crystal to settle before proceeding.

To support fast operation of the serial port, the example selects the faster external clock for both the CPU and for the UART module. The UART

module is also configured to the desired options and baud rate. The details of these settings are found in the UART chapter of the MSP User's Guide. The serial peripheral supports several different interfaces, so the module enable register ME1 is then used to select UART mode. The interrupt enable for the UART receive is then set. Because the serial port uses general I/O port 3, the special pin use must be selected via the Port 3 mode select (P3SEL). The transmit pin direction must be changed to an output. Finally, the last step is to enable interrupts which sets the general interrupt enable (GIE).

With everything configured, the actual interrupt service routine only needs to process the characters received. The interrupt occurs when a character has been loaded from the serial bus into the receive buffer.

More detailed information on the operation of interrupts can be found in the MSP User's Guide. Unfortunately the material is generally found in the chapter for each subsystem. The general interrupt information is found in chapter 2.

Lab 5: Interrupts

Exercise:

Problem:
Seven Segment Counter

Write a function, `void seg_count(int num)`, similar to the LED counting function, `led_num()`, from Lab 4. In this case, the function will display a number between 0 and 15 on the seven-segment display in hex.

The pin-outs for the blue MSP430F16x Lite Development Board are configurable through a programmable PLD. This allows the user to change the pin map in software. You can see the current arrangement of the pins here: LiteBoardPinout

Using the `seg_count()` function you just wrote. Write a program that does the following:

- Poll for button_1 being pushed. Begin counting up slowly on the seven-segment display. It must be slow enough that it can be human readable. Counting will pause if button_1 is pushed again.
- Poll for button_2 being pushed. If it is pushed, begin counting in the opposite direction. Initially, the counter will go up from 0 to F.
- If the counter is going up and it reaches its maximum value, it will roll over to zero and continue counting. If it is counting down it will roll over to its maximum value.
- Be sure to include all appropriate header files, disable the Watchdog Timer, and initialize the master clock.

Exercise:

Problem:
Firing an Interrupt

Interrupts, as the name implies, will interrupt a program whenever it is, and when the interrupt is done being serviced it will jump back to wherever it was in the program. This allows the programmer to have more freedom in their code, automates certain functions, and reduces the number of checks handled by the processor.

In order to fire an interrupt, the following must be done:

- All interrupts on the processor are disabled by default and must be first enabled globally.
- Next, each specific interrupts must be enabled.
- Finally, for each enabled interrupt, an interrupt service routine (ISR), the interrupt handler, must be written. Once an interrupt fires, the program will jump to this section of the program.

A typical program using interrupts will look like the following piece of code: `void main(void){ _EINT(); //Enables interrupts globally //... more code } void SOME_interrupt (void) __interrupt[SOME_vector]{ //...more code that runs once interrupt fires return; }` Processors are designed to be able to service interrupts from predefined, specific places. The list of possible interrupts is put into an interrupt vector, and each must be enabled individually. To find the name of the interrupt vector that needs to be enabled, check the **MSP430Fx16x.h** header file's `Interrupt Vectors` section.

Write an interrupt that will fire when the buttons are pushed. If you need additional help, there are examples in the `../Crossworks MSP430 1.x/samples/` directory.

Modify the program from Problem 1 so that it is interrupt based and does not poll for the buttons being pushed. How does your program behavior changes? When would you want to use polling over using an interrupt?

Timers on the MSP430

The Timer A and B systems on the MSP are a versatile means to measure time intervals. The timers can measure the timing on incoming signals or control the timing on outgoing signals. This function is necessary to meet arbitrary timing requirements from outside components, and the ability is useful in phase locking scenarios etc.

The most basic operation of the timer systems is that the counter in the module counts upward for each clock cycle. The timer clocks can be sourced from both ACLK and SMCLK or from two special external sources INCLK or TBCLK. These last two use general I/O, but allow for a clocking speed not based on the others. The incoming source can be divided down by multiples of two before entering the counter. The user's guide for the MSP has good diagrams on the architecture of the system in the respective sections for Timers A and B. Below the features and uses of Timer B are outlined.

Timer B Modes

Timer B is a flexible system. In addition to clock source selection and input clock dividing, the main counter itself can be set to count to 8, 10, 12 or 16 bits. The main counter can also vary its counting pattern among several options with the MCx bits of the Timer B Control Register TBCTL. These modes are:

1. Stop- the counter is not running
2. Up- the counter counts upward from zero to the value in Timer B Compare Latch 0 (TBCL0). When it gets to this value, it resets to zero. If the TBCL0 value is larger than the allowed maximum number of bits for Timer B, the counter behaves as if in Continuous mode.
3. Continuous- the counter counts from zero to the maximum value specified by the CNTLx bits of the TBCTL register. When the counter reaches this value, it resets to zero.
4. Up/down mode – the counter counts up to the value in TBCL0 then counts back down to zero (as opposed to resetting directly to zero). If

the TBCL0 value is larger than the allowed maximum number of bits for Timer B, the counter behaves as if in Continuous mode.

Please note that the word count is used above- if the timer is set to a certain number it will not trigger anything. The timer must count from the number below to the target number to trigger effects.

Timer B Capture Compare Register 0

There are seven total capture compare registers in Timer B. While there is only one counter for all 7 modules, they can each interpret the count independently. The most important module is module 0 because it controls the timer with its TBCL0 register. Primarily it controls rollovers, but it also has its own dedicated interrupt. Setting this module up correctly is essential for desired operation of Timer B.

What is Capture/Compare?

A capture is a record of the timer count when a specific event occurs. The capture modules of the timers are tied to external pins of the MSP. When the control registers of timer B and the specific capture compare module have been properly configured, then the capture will record the count in the timer when the pin in question makes a specific transition (either from low to high or any transition). This capturing event can be used to trigger an interrupt so that the data can be processed before the next event. In combination with the rollover interrupt on Capture module 0, you can measure intervals longer than 1 cycle.

A compare operation is less intuitive than the capture, but it is basically the inverse of a capture. While capture mode is used to measure the time of an incoming pulse width modulation signal (a signal whose information is encoded by the time variation between signal edges), compare mode is used to generate a pulse width modulation (PWM) signal. When the timer reaches the value in a compare register, the module will give an interrupt and change the state of an output according to the other mode bits. By updating the compare register numbers, you change the timing of the signal level transitions.

This may sound somewhat complicated, but the basic concept of measuring (input) or controlling (output) the time interval between high to low and low to high transitions is all you need to know to start with. The MSP capture/compare modules have many different ways to perform each operation. This can be somewhat overwhelming, but it allows the microprocessor to handle inputs from a greater variety of other components. Capturing and comparing are done with the same modules, and each one can be configured individually. They can also be grouped using the TBCTL to trigger the capture compare registers to load simultaneously (useful for compare mode). The MSP430 User's Guide fully details the behavior of the modules and the registers that control them.

Timer Interrupts

There are two interrupts related to timer B. One interrupt is dedicated to capture compare module 0; and, depending on configuration, it fires when the counter rolls back to zero. The second interrupt handles all 6 other capture compare registers, and fires to indicate that the module has captured or compared as explained above. Each module can be individually masked or enabled, and a special register stores which module caused the interrupt. As with all maskable interrupts, setting the general interrupt enable is necessary to receive them. The interrupts are important in being able to perform complex operations as new data arrives.

Watchdog Timer

What is the Watchdog Timer?

Software stability is a major issue on any platform. Anyone who uses software has probably experienced problems that crash the computer or program in question. This is also true of embedded programs, and in most cases there is no user around to reset the computer when things go wrong. That job is occupied by the watchdog timer. The watchdog timer is a 16 bit counter that resets the processor when it rolls over to zero. The processor can reset the counter or turn it off, but, correctly used, it will reset the processor in case of a code crash. To avoid getting reset, the program must reset the timer every so often. A program which has crashed will not do so, and the system will reset. To improve its efficacy, the watchdog timer register also requires a password. In order to change the lower part of the watchdog control register, the upper part of the register must be written with a specific value. This value is specified by the alias WDTPW in the MSP header files. This password reduces the likelihood that a random crashed instruction could prevent the reset.

Other uses for the Watchdog Timer

In situations where a system crash is not a concern, the watchdog timer can also act as an additional timer. The watchdog timer can be configured to give an interrupt when it rolls over; this interrupt could also be used to handle system crashes. While the watchdog timer is not as versatile as the other MSP430 timers, the watchdog control register WDTCTL still allows selection of the timer’s divider and clock source. Often the watchdog timer is simply turned off by setting the hold bit in the control register. Any changes to this register require writing the password to the upper bits.

Lab 6: Timers on the MSP430

In this lab, we will cover the timing options for the MSP430. The first part explains the clocking system of the processor and the options it allows. The second part will cover the timers and timer interrupts available on the MSP. Each of these sections is strongly related to real time programming, but that topic will be dealt with separately in another lab. In general, a real-time application is one which responds to its inputs as fast as they happen. The microprocessor is generally expected to take less time to handle an interrupt or stimulus than the time before the next event will happen.

The timer system is broken into three primary parts on the MSP430: Timer A, Timer B, and the Watchdog timer. Timer B is larger and more versatile than Timer A. The User's Guide and data sheet will explain the differences, but the way that the control registers configure each timer is largely the same. The watchdog timer will be covered in a module of its own. The timers are closely tied with real time applications because they govern the occurrence of the periodic functions of the processor.

Exercise:

Problem:
Timer A

Refer to **Chapter 11: Timer_A** of the User's Guide to get a detailed description of all the options in Timer A. Basically, setting up the timers requires that you define a source for the timer and to specify a direction for the count. It may also be helpful to clear the timer register before you begin to guarantee and accurate count from the first instance. Set up Timer A in Continuous Mode and sourced from SMCLK. Set TACCR0 and TACCR1 to have two different values. Output TA0 and TA1 from header J8 of the board so that you may directly observe the output of Timer A. You may need to remove the jumpers in order to have access to these signals.

Using two different channels of the Oscilloscope try to recreate parts of **Figure 11-13. Output Example- Timer in Continuous Mode**. On Channel 1 show **Output Mode 4: Toggle** and on Channel 2 show **Output Mode 6: Toggle/Set**. Vary the TACCTLx in order to get as

close to the orginal figure as possible. Take a screenshot of the scope and include it in your lab report.

Try this again using Timer B. This time on Channel 1 show **Output Mode 4: Toggle** and on Channel 2 show **Output Mode 2: Toggle/Reset**. Take a screenshot and submit it. What are the differences between Timer A and Timer B? What was the frequency your signals? What is the relationship between TACCTLx and the frequency?

Exercise:

Problem:
Timer

Set up the timers to fire interrupts. Using the `seg_count()` function from lab 5, use the timers to write a counter. The seven-segment display should display the number of seconds that have elapsed since the timer was started. Button_1 should start/stop the timer. Once the seven-segment gets to its maximum value of 15 it should roll over. There should be no for-loops in your program, and should be entirely interrupt driven. You may use Timer A or Timer B. It is possible to have each Capture Control Register to fire an interrupt once it reaches its max value. How was the timer set up to calculate one second?

Exercise:

Problem:
Duty Cycle

We have discussed earlier that the Duty Cycle related to the width of a pulse. If we trigger an LED with a reletively high frequency square wave, it will appear to be on constantly eventhough it is actually switching on and off quickly. As we vary the duty cycle of our trigger signal, then the LED may appear to get dimmer or brighter depending on which way we vary it.

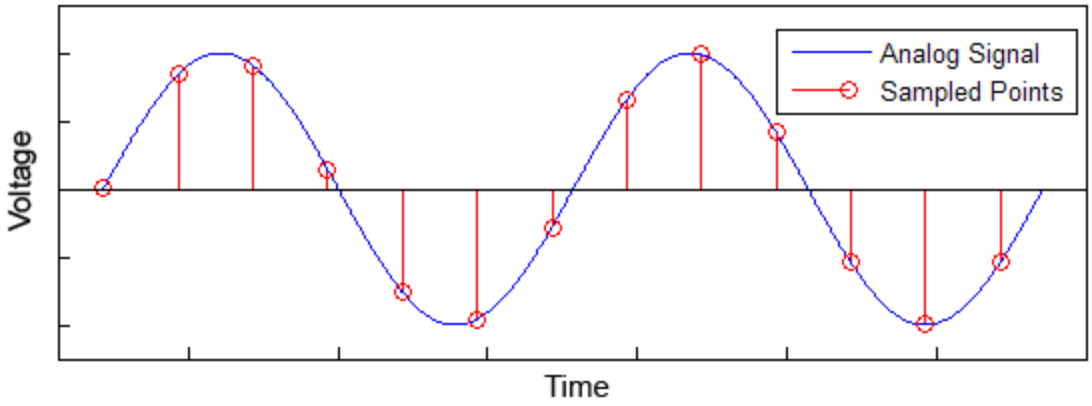
Set up the timers to toggle an LED. Without changing the frequency of your timing pulse, change the duty cycle so that the LED appears to fade in and out. The time it takes to go from completely off to max

brightness shouldn't take more than a second, then it should repeat. Once again, there should be no for-loops in your program, and you can use any combination of the timers that you wish.

Extra Credit: Once you get a single light to fade in and out, try to get all of the lights to fade asynchronously. This means that while one LED is fading out the next one should begin to fade in and so on. Good luck.

Introduction to Sampling

Sampling refers to the process of converting a continuous, analog signal to discrete digital numbers. Typically, an **Analog to Digital Converter** (ADC) would be used to convert voltages to a digital number corresponding to a certain voltage level. The process may be reversed through a **Digital to Analog Converter** (DAC).



This shows the way that a given analog signal might be sampled. The frequency at which the signal is sampled is known as the **sampling rate**.

Resolution

The number of bits used to represent a sampled, analog signal is known as the resolution of the converter. This number is also related to the total number of unique digital values that can be used to represent a signal.

For example, if a given ADC has a resolution of 10 bits, then it can represent 4,096 discrete values, since 2^10 = 4,096.

We may also think about resolution from an electrical standpoint, which is expressed in volts. In that case, the resolution the ADC is equal to the entire range of possible voltage measurements divided by the number of quantization levels. Voltage levels that fall outside the ADC's possible

measurement range will saturate the ADC. They will be sampled at the highest or lowest possible level the ADC can represent.

For example:

- Full scale measurement range: -5 to 5 volts
- ADC resolution 10 bits: 2^10 = 1,024 quantization levels
- ADC voltage resolution: (5-(-5))/1024 = 0.0098 volts = 9.8 mV

Large ranges of voltages will fall into in a single quantization level, so it is beneficial to increase the resolution of the ADC in order to make the levels smaller. The accuracy of an ADC is strongly correlated with its resolution however; it is ultimately determined by the **Signal to Noise Ratio** (SNR) of the signal. If the noise is much greater relative to the strength in the signal, then it doesn't really matter how good or bad the ADC is. In general, adding 1 more bit of resolution is equal to a 6 dB gain in SNR.

Sampling Rate

Analog signals are continuous in time. In order to convert them into their digital representation we must sampled them at discrete intervals in time. The interval at which the signal is captured is known as the **sampling rate** of the converter.

If the sampling rate is fast enough, then the stored, sampled data points may be used to reconstruct the original signal exactly from the discrete data by interpolating the data points. Ultimately, the accuracy of the reconstructed signal is limited by the quantization error, and is only possible if the sampling rate is higher than twice the highest frequency of the signal. This is the basis for the **Shannon-Nyquist Sampling Theorem**. If the signal is not sampled at baseband then it must be sampled at greater than twice the bandwidth.

Aliasing will occur if an input signal has a higher frequency than the sampling rate. The frequency of an aliased signal is the difference between the signal's frequency and the sampling rate. For example, a 5 kHz signal sampled at 2 kHz will result in a 3 kHz. This can be easily avoided by

adding a low pass filter that removes all frequency higher than the sampling rate.

Analog-to-Digital Converter on the MSP430

The analog to digital converter (ADC) on the MSP430F169 is a 12 channel, 12 bit converter. The module is highly configurable and can run largely free of program involvement. In this portion of the lab, we will broadly explain the features of the module, but the particular effects of each register are listed, as usual, in the **User's Guide**.

Range of Measurement

The result of each conversion will be 12 bits long in the form of an unsigned integer whose value is: 4095x (Vin – Vrneg)/ (Vrpos – Vrneg) Where Vin is the input voltage to be measured, Vrneg is the lower reference voltage, and Vrpos is the higher reference voltage. The reference voltages are set to power and ground by default (3.3V and 0V), but they can be changed to several other possibilities using the ADC12 Conversion Memory Control Registers (ADC12MCTLx). This allows each sample to choose its own voltage references. This register also allows for selection of the input channel for each sample. The highest bit of the register is used for multi-channel sequences. This EOS bit indicates the last sample of a sequence.

Operation Modes of the ADC

The ADC12 has four basic operation modes:

1. **Single channel, single conversion** This mode corresponds to a request by the processor for a single sample from a single channel. Interrupts can still be used to indicate when the conversion is complete. The ADC will write the conversion to the ADC12MEMx cell indicated by the CSTARTADDx bits.
2. **Single channel, repeated conversions** This mode uses a single ADC12MEMx cell as indicated by the the CSTARTADDx bits. Because this mode only uses a single memory cell, the results must be collected after each conversion. The interrupt flag is set after each conversion.

3. **Multiple channels, single conversion each** A sequence is set up using the ADC12MCTLx registers to configure each memory slot to sample with the desired parameter. Each cell will take one sample before the sequence will need to be reinitiated. An interrupt flag will be set after each conversion.
4. **Multiple channels, repeated conversions** A sequence is set up using the ADC12MCTLx registers to configure each memory slot to sample in the desired way. The sequence will repeat with the interrupt flag being set after each sample.

For each mode, a complete state machine diagram of the procedure is shown in the **User's Guide** (chapter 17). The particular mode is chosen via the CONSEQx bits of the ADC12 Control Register 1 (ADC12CTL1). The conversions are generally started by setting the ADC12 Start Conversion bit (ADC12SC) or ADC12 Control Register 0 (ADC12CTL0).

Digital-to-Analog Converter on TI MSP430

The MSP430F169 is equipped with two digital to analog converters (DACs) on general I/O port 6. A DAC converts an unsigned integer into an analog voltage within the capacity of the MSP. As in all cases with the MSP, the complete details are found in the MSP430 Family User's Guide from the Texas Instruments website www.ti.com.

Voltage Range and Precision

Unlike the ADC on the MSP, the DACs have variable bit precision of either 8 or 12 bits. The range is also controllable as either one or three times the voltage reference. The voltage reference is also selectable. All bits for controlling voltage reference, scaling, and precision can be found in the DAC12 Control Register (DAC12_xCTL). In most cases, 12 bit precision is desirable.

In the MSP430 architecture there are two general sources for a voltage reference: VeRef and Vref. VeRef is an externally supplied voltage, but Vref is generated by the MSP itself. Because the 226 board does not provide the MSP with a VeRef voltage, the only voltage reference available for the DAC and ADC is the Vref. This restricts the choices for the SREF bits of the DAC control register to the Vref choice. Additionally, the Vref voltage reference for the DAC comes from the ADC module. This means that it is necessary to turn on the voltage reference in the ADC control register (ADC12CTL0) with the REFON bit.

Operating Modes

The DAC competes with two of the ADC channels for access to the pins of I/O Port 6. Unlike most peripheral controls on the MSP430, the DAC's access to the pins is not controlled by the PxSEL register. Instead, the DAC is multiplexed with the ordinary port control and will drive the output if the DAC12AMPx bits are set to any value greater than zero. The DAC12AMPx bits in the control registers also determine the mode of the DAC's operation.

- Mode 000 The input buffer is off. The DAC is off, and the output buffer is high impedence.
- Mode 001 The input buffer is off. The DAC is off, and the output buffer is 0V.
- Modes 010 - 111 Each step in mode increases the power consumption of the DAC while improving the settling time of the module.

The two most useful modes for these labs are modes 000 and 111. Because our devices are not battery powered, we can afford the power consumption to improve speed. Mode 000 is needed to turn off the module when necessary.

Using the DAC

The basic method of DAC operation is to load the DAC12_xDAT. Using the DAC12LSELx bits of the control register, the actual trigger for the output is set. The device can be figured to output whenever the data register is written to, whenever the data register is written and the DAC enable is set, or according to timers A or B. It is necessary to change the mode out of SEL mode 0 in order to use the interrupts with the DAC.

The DAC can be interrupt enabled via the DAC control register and the interrupt enable registers. The flag is set whenever a conversion is complete. Also, the DAC outputs can be grouped together to ensure that the output of both DACs appears simultaneously. When the units are grouped, both DAC12_xDAT registers must be written before the outputs will appear. The device does not check whether you have changed the value; updating with the same information will still work.

Lab 7: ADC, DAC, and Mixed Signals

ADC, DAC, and Mixed Signals

Exercise:

Problem:
Quickies

1. The MSP430 has both a 10 and 12-bit ADC. The number of bits used by the ADC is known as its resolution. You may learn more about sampling and Analog to Digital Converters from the Introduction to Sampling module. How many possible values can be represented for each of the 10 and 12 bit cases?
2. Extreme voltages, either too high or too low, cannot be measured correctly. What is the range of analog voltages that can be accurately represented on The MSP430F16x Lite Development Board? You may want to check the User's Guide or experiment with the hardware.
3. In the real world, signals are polluted by "noise" that alters the quality of the original signal. Signal to Noise Ratio, SNR, is often used as a measure of the quality of a signal. Before a signal is sampled through the ADC, it is helpful to condition the signal in order to improve its SNR. What can be done to condition the signal? Where would it be ideal to condition it and why? (i.e. at the ADC, near the source, at the processor?)
4. The Nyquist Theorem states that a signal must be sampled at least twice as fast as the highest frequency in order to be decoded without error. The minimum sampling frequency is therefore known as the Nyquist Sampling Rate. Typical, audio CDs are sampled at a rate of 44.1 kHz, and can be decoded to 65,536 unique voltage levels. What is the highest frequency that can be represented without error? How many bits per sample are used? What is the total data rate for a stereo CD?

Exercise:

Problem:

ADC Setup

1. Figure out what the following codes is doing. Set up the hardware so that it functions correctly, and comment each line of code. What is the code's function? `#include <msp430x16x.h> #define yellow 0x80 #define set_led(led, state) { if (state) P1OUT &= ~(led); else P1OUT |= (led); } void main(void){ P1DIR |= 0x80; set_led(yellow,0); WDTCTL = WDTPW+WDTHOLD; P6SEL |= 0x04; ADC12CTL0 = ADC12ON+SHT0_2; ADC12MCTL0 = SREF_0 + INCH_2 + EOS; ADC12CTL1 = SHP; ADC12CTL0 |= ENC; while (1) { ADC12CTL0 |= ADC12SC; while ((ADC12IFG & BIT0)==0); if (ADC12MEM0 >= 1620) set_led(yellow,1) else set_led(yellow,0) _NOP(); } }`

> **Note:**Most modern compilers intended for use with embedded processors, such as the Rowley's CrossWorks for MSP430, allow the user to check the status of the registers while the program is halted. This is extremely helpful in debugging code. For example, if the program is halted with a `NOP()` after a sample is taken from the ADC, the user may check the `ADC12MEMx` register to see the new value that has been stored. If a value has changed since the last time the processor was halted, it will turn red.

2. Create a version of the program that is interrupt driven and uses **Repeat-single channel Conversion Mode**. The original program uses a while-loop to poll the interrupt flag. What is the sampling rate?
3. Using **Repeat-sequence-of-channels Conversion Mode**, write an interrupt driven program to sample analog input channels 1

and 2. As before, toggle an LED for each channel as it passes the 1.5V threshold. You should now have two separate analog voltages controlling two separate LEDs.

Exercise:

Problem:
DAC Setup

1. Configure that `DAC12_1CTL` register so that DAC0 outputs a triangle wave. This program should be interrupt driven, and any computation of the triangle wave should be in the ISR. View the output on the oscilloscope and take a screenshot.

Exercise:

Problem:
Stereo to Mono Mixer

1. Combine the last two problems to create a stereo to mono mixer. The program should sample ADC0 and ADC1, add the two signals together, and output the result from DAC1. It is possible to write the contents of `ADC12MEMx` directly to `DAC12_xDAT`. You should also scale down each of the input signals so that you don't saturate the output. Only use a single interrupt. Take a screenshot with a) your stereo input signals (make sure that they are different) and b) your mono result. Could you process an audio CD like the one described in Problem 1.4? Explain.

Real Time

What is Real Time?

The term real time refers to systems whose ability to process data and commands is not only fast enough so as to seem instantaneous (relative to users or systems that rely on it) but who can handle as much data or as many requests as the users and systems are expected to provide. These two standards for "real time" can be described by as "bandwidth" and "worst case response."

Often, high bandwidth and low latency go hand in hand because expectations of quality in a system extend to both responsiveness and capacity. Three examples: the real time computer game, the cell phone, and the control systems of a missile.

Bandwidth

Bandwidth is a measure of the amount of data a system can handle on average. Bandwidth is expressed in hertz or bits per second. "Bits per second" is the term generally used for computer systems processing data. Hertz is the term generally used for communications and analog systems. Actually, the terms refer to the same unit because a "bit" has no unit. The more bandwidth a device has, the more data it can process per second or the larger the range of frequencies it can support. If a device does not have sufficient bandwidth, then there are two common ways in which the system can handle the problem.

If the shortage of bandwidth is only temporary, due to a spike in requests or short term complication, then a system may support queuing the data for later processing. When the input has returned to "normal," the system can use excess bandwidth to catch up. Being able to support queuing is expensive because extra memory and bandwidth will be required.

If the shortage of bandwidth lasts longer than the system can handle, or if the system cannot handle a shortage at all, then the system will lose the

extra bandwidth. This can mean that frequencies coming into the system will not come out again, or this can mean that some (or all!) of the requests will be ignored.

Some systems are designed so that they will always have sufficient bandwidth. This is more common for real time systems. Other systems are designed to have enough bandwidth "on average." The internet is a major example of such a system. In general, ensuring that a design has sufficient bandwidth for the desired application is a major concern of an engineer.

For the first example, the real time computer game, bandwidth corresponds to the maximum amount of input the game can handle from the internet, keyboard, mouse, etc without skipping, losing commands, or generally messing up. Because modern computers generally far exceed the user's ability to provide input, this is rarely a problem.

With cell phones, the device is specifically designed to support a known bandwidth of communication, a two-way voice conversation. Because the input bandwidth is very stable, the cell phone itself does not have latency or dropping problems. Problems do occur between the cell phone and the cell phone tower, and this can cause skipping in the conversation. These problems are more related to the random noise inherent to wireless communication than to the insufficient capacity of the cell phone. The telephone system transmits sound information with a bandwidth of a few thousand hertz, so the bandwidth of the cell phone must also be able to provide this.

Finally, the control systems that allow guided missiles to fly long distances at great speed to hit very precise targets must guarantee that the information necessary for these on-the-fly calculations is available as fast as it is needed. For situations with no tolerance for error, bandwidth needs to be sufficient for all situations. The missile designer will avoid using systems that "statistically" provide enough bandwidth. The data paths will need to be guaranteed to be sufficient for the application at all times.

Worst Case Response and Latency

Latency is a word commonly used for the delay between the time the input enters a system and the time the output leaves. A real time system is a system that can guarantee that no input's latency is longer than a specified maximum. Not only must the statistical average latency be less than the specified maximum, but the worst case latency must also meet this requirement for strict real time operation. To calculate a worst case latency, the absolute slowest path from the input to the system through the hardware and software must be added together. This means that software conditions must be examined for the entire logical path through the software. Generally the "specified maximum" is dictated by the maximum delay that can be tolerated by the user.

In a real time computer game problems with latency result in a noticeable delay between the time when the mouse is moved and the screen updates. In computer games this happens because the system has more things visible on the screen than it has time to calculate the graphics for before the screen refreshes. While lag is annoying, players can tolerate occasional problems.

In cell phone communications, there is much less tolerance for latency. In general, cell phone users expect for cell phones to be as instantaneous as land line phones. Therefore, the cell phone system will generally throw away any part of the conversation which is delayed too much rather than play the sound late. This way only gaps in the connection rather than sound delays will be heard on the other end. As with bandwidth, the cell phone designer designs the cell phone so that the hardware system will be able to reliably get one sound sample to the other end of the connection in time for the delay not to be noticed by human perception.

Finally, missile systems have far less tolerance for latency than the other applications. In high precision, high reliability systems the worst case must be taken into account because any single failure to meet latency requirements can be disastrous for the entire system. Furthermore, the high speeds of aerospace applications mean that there is less absolute time for calculations to be made.

What is Direct Memory Access (DMA)?

Direct memory access is system that can control the memory system without using the CPU. On a specified stimulus, the module will move data from one memory location or region to another memory location or region. While it is limited in its flexibility, there are many situations where automated memory access is much faster than using the CPU to manage the transfers. Systems like the ADC, DAC and PWM capturing all require frequent and regular movements of memory out of their respective systems. The DMA can be configured to handle moving the collected data out of the peripheral module and into more useful memory locations (like arrays). Only memory can be accessed this way, but most peripheral systems, data registers, and control registers are accessed as if they were memory. The DMA is intended to be used in low power mode because it uses the same memory bus as the CPU and only one or the other can use the memory at the same time.

The DMA system is organized into three largely independent parts. Though the three compete for the same memory bus, they have can be configured for independent triggers and memory regions.

DMA Operation

There are three independent channels for DMA transfers. Each channel receives its trigger for the transfer through a large multiplexer that chooses from among a large number of signals. When these signals activate, the transfer occurs. The DMAxTSELx bits of the DMA Control Register 0 (DMACTL0). The DMA controller receives the trigger signal but will ignore it under certain conditions. This is necessary to reserve the memory bus for reprogramming and non-maskable interrupts etc. The controller also handles conflicts for simultaneous triggers. The priorities can be adjusted using the DMA Control Register 1 (DMACTL1). When multiple triggers happen simultaneously, they occur in order of module priority. The DMA trigger is then passed to the module whose trigger activated. The DMA channel will copy the data from the starting memory location or block to the destination memory location or block. There are many variations on this,

and they are controlled by the DMA Channel x Control Register (DMAxCTL):

1. **Single Transfer** - each trigger causes a single transfer. The module will disable itself when DMAxSZ number of transfers have occurred (setting it to zero prevents transfer). The DMAxSA and DMAxDA registers set the addresses to be transferred to and from. The DMAxCTL register also allows these addresses to be incremented or decremented by 1 or 2 bytes with each transfer. This transfer halts the CPU.
2. **Block Transfer** - an entire block is transferred on each trigger. The module disables itself when this block transfer is complete. This transfer halts the CPU, and will transfer each memory location one at a time. This mode disables the module when the transfer is complete.
3. **Burst-Block Transfer** - this is very similar to Block Transfer mode except that the CPU and the DMA transfer can interleave their operation. This reduces the CPU to 20% while the DMA is going on, but the CPU will not be stopped altogether. The interrupt occurs when the block has completely transferred. This mode disables the module when the transfer is complete.
4. **Repeated Single Transfer** - the same as Single Transfer mode above except that the module is not disabled when the transfer is complete.
5. **Repeated Block Transfer** - the same as Block Transfer mode above except that the module is not disabled when the transfer is complete.
6. **Repeated Burst-Block Transfer** - the same as Burst Block Transfer mode above except that the module is not disabled when the transfer is complete.

Writing to flash requires setting the DMAONFETCH bit. If this is not done, the results of the DMA operation are “unpredictable.” Also, the behavior and settings of the DMA module should only be modified when the module is disabled. The setting and triggers are highly configurable, allowing both edge and level triggering. The variety of settings is detailed in the DMA chapter of the users guide. Also, it is important to note that interrupts will not be acknowledged during the DMA transfer because the CPU is not active. Each DMA channel has its own flag, but the interrupt vector is

shared with the DAC. This necessitates some software checking to handle interrupts with both modules enabled.

Lab 8: DMA + RS232

Direct memory access is a system to transfer data between peripheral modules and memory without using processor instructions. While its operation does occupy the memory bus, far fewer instructions require CPU interaction. This allows the CPU to work on other tasks simultaneously, or it may be be put into a low power mode.

Exercise:

Problem:
DMA Powered Voltmeter

The DMA module allows you to automatically move data between memory locations. This will also allow us to automate much of your program's execution.

1. Set up the DMA module to automatically transfer the data from the ADC to the DAC. Transfer the input from the ADC1 to DAC0 and ADC0 to DAC1. Sampling should be interrupt enabled.
2. Measure the peak-to-peak amplitude of the input signal to ADC0 using the processor. Units are irrelevant. Display the amplitude on the seven segment display. You should have at least 4 different possible levels.

Exercise:

Problem:
RS232

We will configure the serial port (RS232) to transmit data. Just like before create a new project and do the following: disable the watchdog timer, and initialize the master clock and i2c. This should now be standard for all new projects. To set up the UART to use RS232 we must set the following registers:

- Reset UICTL (on its own line of code)
- Set the character length to 8-bits
- Set U1TCTL to enable ACLK as the BRCLK clock.

- The baud rate should have the following settings: `U1BR0 = 0x7c;` and `U1BR1 = 0x01;` This will set the baud rate at 19,200 bits per second.
- Set ME2 to enable both transmit and receive.
- Enable UART_TX and UART_RX on the msp (Hint: use P3SEL)
- Set UART_TX as an output pin.

To transmit data, check if the UTXIFG1 flag is set in UTXIFG1. Once the flag is set you may write directly to TXBUF1, and that data will be transmitted. Now, come up with an array of ASCII characters in hexadecimal notation, and load each character in the TXBUF1 one at a time. Don't for get to add a new line character, `/n`, at the end of your message.

To test your project open up HyperTerminal, and set the correct baud rate (19200). All other default settings should be fine. If you correctly configured the UART then you should see your message in the terminal.

Once you have verified that you can transmit data to the serial port, load the `TXBUF1` via DMA. You may trigger the DMA any way you wish. This will eliminate most of the work that the processor has to do.

Exercise:

Problem:

Modify the first problem so that you input something into ADC0 and DMA the samples to the serial port. Come up with some way for a computer to read those values, HyperTerminal or otherwise.

Memory Conservation

In the early days of computers, the instruction memories of main frames were incredibly small by today's standards - in the hundreds or thousands of bytes. This small capacity placed the emphasis on making each instruction count and each data value saved useful. Fortunately, just as processors have become millions of times faster, program memory have become similarly increased. However, there are still general practices that must be kept in mind when using program memory. Further, smaller platforms like microcontrollers are still limited to program and data memory in the kilobytes. This module explains the kinds of memory, some common memory organizations, and the basics of conserving memory.

How memory is organized

So far in this module, we have referred to the program memory of a computer (the RAM of a PC), but in most memory architectures there is some categorization of the memory into parts. The basic principle behind subdividing the memory is that by breaking the memory into sections, it will be easier to access the smaller memory. Also, clever memory restrictions allow the designer of the system to improve performance. Strict divisions between memory sections are also very important for compilers to be able to utilize the memory.

Instruction memory is a region of memory reserved for the actual assembly code of the program. This memory may have restrictions on how it can be written to or accessed because it is not expected that changes will need to be made frequently to the code of the program. Because the size of instruction memory is known when the program compiles, called compile time, this section of memory can be segmented by hardware, software, or a combination of the two.

Data memory is a region of memory where the temporary variables, arrays, and information used by a program can be stored without using the hard disk or long term memory. This is the section memory that memory allocations come from when more memory for data structures is needed in the course of the program.

Heap memory is an internal memory pool that tasks use to dynamically allocate memory as needed. It may be used when functions must be put on hold and the function's data needs to be stored. As functions call other functions, it is necessary that the new (callee) function's data be loaded into the CPU. The previous (caller) function's data must be stored in the heap memory. The deeper function calls go, the larger the heap portion of memory needs to be.

Often, the heap memory and the data memory compete directly for space while the program is running. This is because both the depth of the function calls and the size of the data memory can fluctuate based on the situation. This is why it is important to return the heap memory the task uses to the memory pool when the task is finished.

Memory Allocation in Languages

The organization of memory can vary among compilers and programming languages. In most cases, the goal of memory management systems is to make the limited resource of memory appear infinite (or at least more abundant) than it really is. The goal is to free the application programmer from having to worry about where his memory will come from. In the oldest days of mainframes, when each byte of memory was precious, a programmer might account each address in memory himself to ensure that there was enough room for the instructions, heap, and data. As programming languages and compilers were developed, algorithms to handle this task were developed so that the computer could handle its own memory issues.

Today, even assembly programmers do not have to worry about memory allocation because the assembler will handle that. The memory allocation programs are good enough at solving the problem that it isn't worth a programmer's time to solve this problem. Instead, there are a few different ways that languages solve the problem of memory allocation. In general, it is a simple matter to provide the programmer with the memory that is known to be needed at compile time. This includes primarily global data values and the space for the code itself. The more difficult problem is how

to provide flexible data memory that may or may not be needed when the program actually executes.

The approach that C takes is to ask the programmer to call special functions that manage memory allocation. These methods are called `malloc(int)` and `free(void *)` The basic idea is that whenever the program needs a specific amount of additional memory, it calls malloc (memory allocate) with the integer being the number of bytes of memory needed. The program will then search for a block of memory of the appropriate size and return a pointer to it. When the program is done with a particular allocation of memory, it calls free to let the memory management library know about the particular block of memory isn't needed anymore. If the programmer is diligent about returning (freeing) memory that isn't needed anymore, then the programmer will enjoy abundant memory without having to count individual bytes. On the other hand, if a program repeatedly requests memory but does not free the memory back to the system, the memory allocator will eventually run out of memory. The program will then crash. Thus, it is essential for passages of code that frequently request memory allocations to free these allocations as they can. Un-freed allocations are not fatal in very infrequently executed parts of code; however, the longer a program runs, the more the problem will accrue. In general, a program that allocates but does not free memory, gradually using unnecessarily more memory over time, is said to have a **memory leak**.

Other languages handle the problem of memory allocation automatically. Java will allocate the memory for new data on the fly using the keyword new instead of the function malloc, but the more important difference is that freeing takes place automatically. Part of the Java system called the **garbage collector** detects memory that can be safely freed and does so. In this fashion, Java programs do not suffer memory leaks in the way a C program might.

Memory and the MSP

In the MSP430 there is no inherent difference between instruction memory, data memory, and heap memory. The only subdivisions in memory are the blocks of flash and the section of RAM. Any of these sections can hold

instructions or other kinds memory. However, because it is problematic to erase and rewrite flash in the middle of program execution, the flash memory is best saved for instructions and constants. The remaining RAM must be shared between the heap, the dynamically allocated memory, and the global variables. On the MSP430F169, there is only 2KB of RAM, so no memory leaks are tolerable.

How memory is wasted or conserved

Memory leaks, the most notable way to waste memory, have already been discussed, but there are several others. While memory leaks abuse the dynamically allocated portion of data memory, many layers of function calls abuse the heap. Above, it was explained that each time a function calls another function, the caller's registers and data are moved onto the heap. If each called function calls another function in turn, then the heap portion of the memory will grow significantly. For high power computing systems, this is not usually a great threat to the overall supply of memory compared to memory leaks. Embedded systems however must avoid deep layers of function calling or risk exhausting the overall supply of memory.

There is also a programming technique called recursion wherein a recursive function calls itself repeatedly on progressively smaller or simpler versions of the data until the answer is trivial. While this technique leads to some very clever solutions to some complex problems, it is uses large amounts of memory to achieve this end. Therefore, recursion is generally a poor choice when memory must be conserved.

Finally, another important way to waste memory is to create too many global variables. Specifically, variables whose scope could be local or who could be allocated dynamically waste memory because they take up space even when they aren't being used. Use malloc and free to avoid using as many global variables.

Improving Speed and Performance

So far in this course, programming assignments have focused on functionality. In most applications of embedded programming, speed and power performance are equally important. Long battery life is won through judicious hardware and software design. Skillful programming will allow the same job to be done with cheaper parts to improve the bottom line. This lab will introduce the basic concepts behind power and speed performance improvement.

Speed Performance

It is well known from the consumer PC market that the speed of computers can be measured in hertz. It is less well known that the frequency of the computer's processor does not adequately indicate a computer's performance or even the performance of the processor itself. By using the ELEC226 board, the choice of processor and speed has been made, but the question of speed is a different one in embedded programming. While the dominant paradigm in consumer personal computing is to increase the performance of the computer to allow the system to do more with each generation, embedded processors are chosen to be able to perform specific tasks. The cheapest processor that can meet the specifications for the design will be chosen. While the issue and business conditions make the situation much more complicated than just price, the pressure is still toward choosing a part with **less** performance, not more.

In order to improve the performance of a software application, it is necessary to understand the way performance is measured. Measuring performance between platforms and software packages is a problematic endeavor, improving the performance of a single program on a single platform is much simpler. a detailed explanation of the nuances of performance measurement in computing is beyond the scope of this lab, simple way to gauge the amount of time a program will take to perform a task is to count the number of processor cycles that the code will take. On the MSP430, each CPU instruction, jump, and interrupt takes a fixed number of cycles as explained in the MSP430 User’s Guide. Taking into

account branching, function calls, and interrupts the assembly code of a program can be used to calculate the time needed for a section of code.

Performance Tips

As mentioned above, embedded programming has different priorities from personal computing. Because the embedded programmer is usually trying to accomplish a specific task within a certain amount of time, the most important test of performance is whether the program is performing calculations on the inputs as fast as the inputs can enter the system. The goal is to make applications "real time."

When the first draft of a program is unable to keep up with the required sampling, it is necessary to reduce execution time. Often, changing the hardware configuration is not possible; and software speed gains are almost always more cost effective.

There are many approaches to improving speed performance. Incredible amounts of research go into new algorithms for common problems to improve the way that problem is solved. However, simply eliminating unnecessary instructions is a good start to improving performance. Test code left in a final version, any unnecessary instructions in a loop, and can all significantly increase the time in a section of code.

In C, unnecessary code takes the form of too many method calls inside of a loop because each function call costs additional instructions. While this is not an important loss for code that is only executed once per sample, in loops that run often, every little gain counts much more. When trying to reduce execution time, it is best to start with the regions of the code where the processor spends the most time. Parts of the program that are only executed rarely have only a small effect on the speed compared to a loop that might run 100 times per sample. If something can be done once, outside of the loop, do not do it many times inside the loop.

Another straightforward way to maximize performance on the hardware provided is to make judicious use of timers and other instruction saving interrupts. The timer interrupts allow the processor to periodically check on

the status of the program without the use of slow while () loops. However, for correct program behavior, it is important to do the minimum possible in the interrupt. This is most important with interrupts that happen frequently because the control flow of the program can be thrown off when interrupts happen faster than the system can handle them. If the same interrupt occurs a second time before the first occurrence of the interrupt has exited there, program behavior is much more difficult to control. It is much easier to simply ensure that the interrupt is short enough to avoid the danger all together.

Avoid recalculating values. If a piece of information is reusable, save it rather than recalculating it to save time. Sometimes memory is so scarce that this may not be possible.

Don’t use the printf function unless you absolutely must. It can be quite slow if used repeatedly. Use breakpoints instead.

Don’t leave legacy code from previous revisions running. If you believe you may no longer need a part of the program, comment it out and note what you did in the comments.

Reducing Power Consumption

One of the most important quality standards for battery powered devices is battery life. Handheld medical tools, electricity meters, personal digital assistants, and a goal of the designer and programmer is to lower the power use of the embedded system to negligible levels. This portion of the lab will give an overview of how power can be conserved using hardware and software. In designing battery powered devices, savings can be gained from the choice of electronic components, the arrangement of components, and the software on the design. The exercises will integrate the low power modes of the MSP into existing labs, so that examples of software power savings can be shown.

Measuring power on the 226 board with the MSP

The 226 board has the ability to measure its own current consumption from the USB cable. The ZXCT1009F (component U10) connects to channel 5 of the ADC on the MSP. This voltage will be proportional to the current passing through the device. The circuit can measure up to 500 mA used.

Shutting off parts in general

Most parts have a shutdown or sleep mode available that will reduce the current consumption of the component considerably. In general, digital parts consume significant current when their transistors switch because of the charging and discharging of the internal capacitances of the transistors. Analog integrated circuits also support shutdown modes to reduce power consumption. Datasheets will specify the current consumption in both on and shutdown modes of the component. It is important to note that when a device is in shutdown mode, power and ground voltages are still powered and connected to the device.

In order to shutdown most integrated circuits, all that is required is a shutdown or sleep pin to be asserted properly. Other devices require a shutdown command to be issued over the bus. The primary disadvantages of shutdown modes, apart from the fact that the device is inoperative is that recovering back into normal operating modes can impose a significant

delay. A useful property of the MSP is that its recovery time from some low-power modes is fast enough to meet the response times of interrupts.

Parts without built-in shutdown modes must be shutdown by having its current supply controlled through a transistor or other switching device.

Using the Low Power Modes

The MSP430 was designed with the low power modes in mind from its beginnings. In lower power mode, the processor can achieve current in the microamps while still monitoring its inputs. While the 226 board cannot take advantage of this ability because of the other higher power components on the board, the principles of utilizing the MSP power modes are described in detail in the second chapter of the MSP User's Guide. The modes vary the degree to which the processor is aware of its surroundings and the clocks that the processor keeps running. The processor lowers power consumption partly by shutting off external and internal oscillators.

There are four low power modes in addition to regular operating mode on the MSP430:

- Active Mode is the fully powered mode when the processor executes code and all clocks and peripherals are active. The chip consumes about 340 µA with 1 MHz clock at 3.3V in this mode.
- Low Power Mode 1 (LPM1) disables the CPU and MCLK while leaving the ACLK and SMCLK enabled. This allows timers, peripherals, and analog systems to continue operation while dropping current consumption to about 70 µA with 1MHz clock at 3.3V. Because the timers and other internal interrupt systems still operate, the processor will be able to wake itself.
- Low Power Mode 2 (LPM2) disables the CPU, MCLK, and the DCO are disabled but the SMCLK and ACLK are active. The DC is disabled if the DCO is not used for MCLK or SMCLK in active mode. Internal interrupts can still operate. Current consumption drops to about 17 µA.
- Low Power Mode 3 (LPM3) disables the CPU, MCLK, SMCLK, and DCO. The DC and ACLK remain active. This allows some peripherals

and internal interrupts to continue. Current consumption drops to about 2 µA.
- Low Power Mode 4 (LPM4) Current consumption drops to about .1 µA, but all clocks and the CPU are disabled. This prevents any of the on-chip modules from operating, and only off-chip interrupts can wake the device.

To enter a low power mode the status register in the CPU must be set to indicate the desired mode. Specifically the bits SCG1, SCG0, OSCOFF, and CPUOFF. The User's Guide details the specific bits needed. Also provided in the chapter is some example code on changing power modes. To exit low power mode, an interrupt is needed. In the interrupt, the previous status register state can be altered so that exiting the interrupt will leave the processor awake. The User's Guide explains in detail the specifics of entering and leaving low power mode. Example code with the compiler also demonstrates the low power modes.

Principles of low power operation on the MSP

The User's Guide for the MSP also explains the principles needed to lower the power consumption of a design. These principles assume that the microcontroller is a significant portion of the board's current consumption. In the case of the 226 board, this is not the case. First, minimize wasteful code execution. This is the same idea as improving speed performance because every unnecessary instruction wastes a little bit of battery power. All of the techniques that improve code efficiency will improve power efficiency. Increasing clock speed will not yield similar power savings because faster execution increases power consumption. Similarly, unused peripheral modules on the processor should be de-activated to save power. Use interrupts to handle events to allow the processor to stay in Low Power Mode 3 as much as possible. By reducing the awake time of the processor, the average current consumption of the MSP can be reduced to levels approximately as low as LPM3 while maintaining the same functionality.

Lab 9: Optimization and Low Power Modes

Low Power Modes and Code Optimization

Exercise:

Problem:
Fibonacci Optimization

The "Reducing Power Consumption" module discusses why it is important to keep power in mind when programming embedded devices. We have yet to consider this while programming the previous labs. Writing efficient code is the first step in improving power consumption, next we can disable all parts of the board that aren't currently being used.

Take the following piece of code: `long fibo(int n) { if (n < 2) return n; else return fibo(n-1) + fibo(n-2); }` It recursively calculate the nth number in a Fibonacci sequence recursively. Recursion makes this piece of code easier to read, however, it is very inefficient and consumes far more memory than it has to. If you try to compute a large number, say `fibo(50)`, then it will take much longer and will consume more power than it should.

The original program is very inefficient and wastes memory in several of the ways described in the inefficient Memory Conservation module. Modify the code to eliminate the memory waste and improve the speed of the program. Note that there is a tradeoff between speed and memory (though at first the program is simply gratuitously wasteful). What is the nature of the tradeoff? Assuming the one addition takes one cycle to complete, how long would it take the original code to complete `fibo(50)`? How long would it take your new, improved version? Assume that you are only considering the addition operations.

Exercise:

Problem:
Low Power Modes

Modify your project so that the processor remains in one of the low power modes whenever it is not doing any calculations. Wake up from low power mode when a pushbutton interrupt fires, and have your program compute `fibo(50)`. Output the result to the standard out display. What is the result? (Hint: 12,586,269,025) Make the result is correct number. As soon as the calculation is done, return to low power mode. Make sure to turn on the Red LED while in an idle state.

Note:A number must be small enough to fit in its given type. If it is too large, you may get unpredictable results. Try using a `long long` for extra huge numbers. If your standard out does not support such large data types then you may have to use bit-wise operations to separate the number into smaller chunks suitable for printing.

Measure the power consumed by the entire device when you are in low power mode and when it is computing something. You may want to have the processor compute something indefinitely, in order to get a more accurate result.

FIR Filters

In signal processing, there are many instances in which an input signal to a system contains extra unnecessary content or additional noise which can degrade the quality of the desired portion. In such cases we may remove or filter out the useless samples. For example, in the case of the telephone system, there is no reason to transmit very high frequencies since most speech falls within the band of 400 to 3,400 Hz. Therefore, in this case, all frequencies above and below that band are filtered out. The frequency band between 400 and 3,400 Hz, which isn't filtered out, is known as the passband, and the frequency band that is blocked out is known as the stopband.

FIR, Finite Impulse Response, filters are one of the primary types of filters used in Digital Signal Processing. FIR filters are said to be finite because they do not have any feedback. Therefore, if you send an impulse through the system (a single spike) then the output will invariably become zero as soon as the impulse runs through the filter.

How to characterize digital FIR filters

There are a few terms used to describe the behavior and performance of FIR filter including the following:

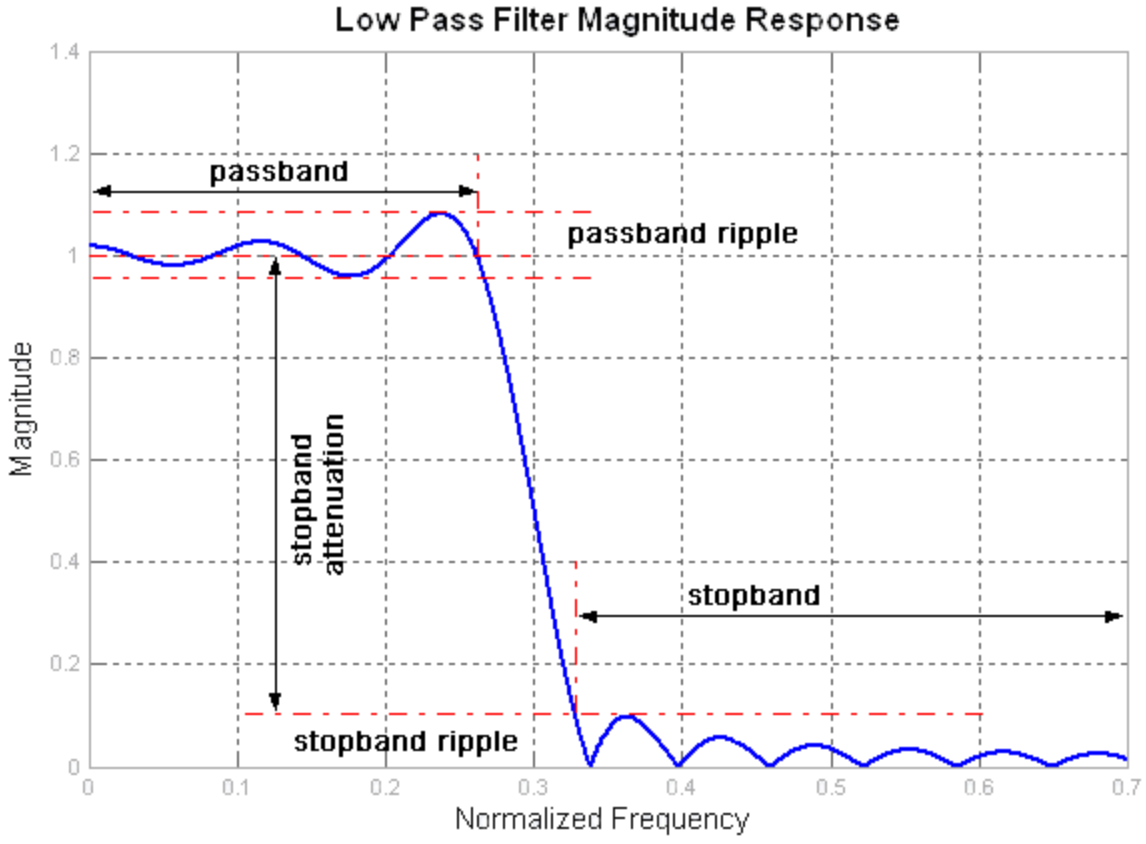
- **Filter Coefficients -** The set of constants, also called tap weights, used to multiply against delayed sample values. For an FIR filter, the filter coefficients are, by definition, the impulse response of the filter.
- **Impulse Response** – A filter's time domain output sequence when the input is an impulse. An impulse is a single unity-valued sample followed and preceded by zero-valued samples. For an FIR filter the impulse response of a FIR filter is the set of filter coefficients.
- **Tap** – The number of FIR taps, typically N, tells us a couple things about the filter. Most importantly it tells us the amount of memory needed, the number of calculations required, and the amount of "filtering" that it can do. Basically, the more taps in a filter results in better stopband attenuation (less of the part we want filtered out), less

rippling (less variations in the passband), and steeper rolloff (a shorter transition between the passband and the stopband).

- **Multiply-Accumulate (MAC)** – In the context of FIR Filters, a "MAC" is the operation of multiplying a coefficient by the corresponding delayed data sample and accumulating the result. There is usually one MAC per tap.

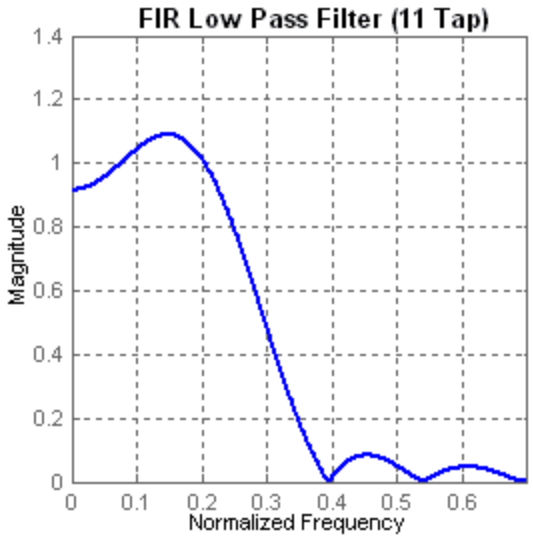
General types of filter

There are a couple different basic filter responses. Each will have a unique frequency response based on its cut-off frequency, the number of taps used, its roll off, and amount of ripple. The various attributes describing a filter may be seen in the following diagram:



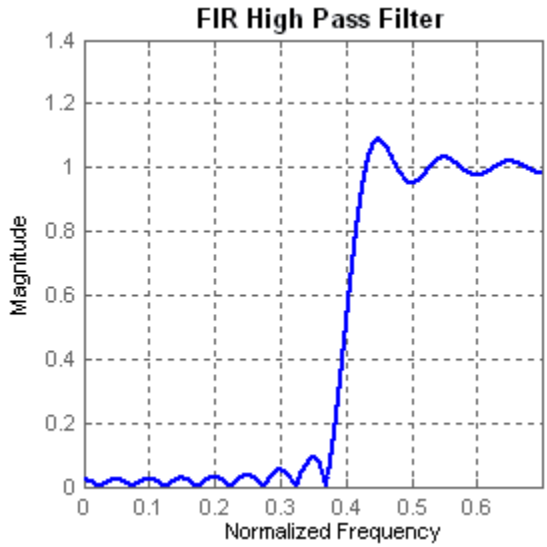
This figure demonstrates an FIR low pass filter with 40 taps.

Reducing the number of taps used in the filter will reduce the number of calculations to process in the signal, however, the quality of the filtering will suffer. Rippling will become more sever, the rolloff will be less steep, and the passband will be less accurate. This may be seen in the following diagram where fewer number of taps were used.

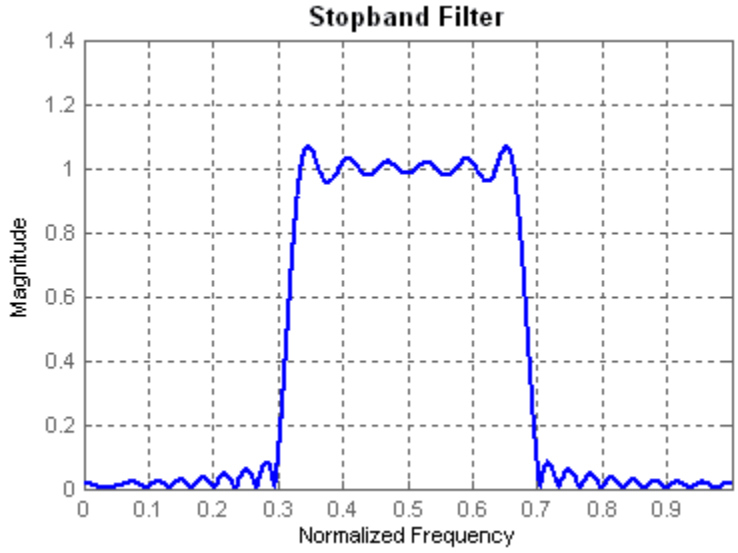


Using only 11 taps has degraded the filter from figure 1.

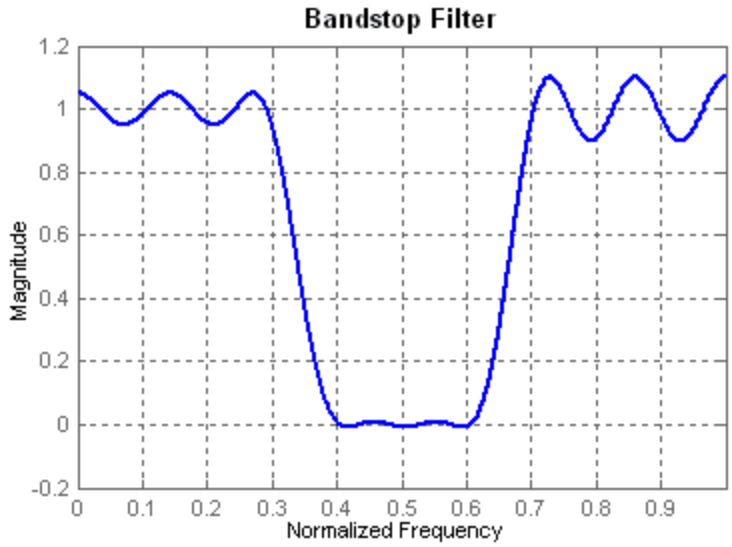
All filters may be categorized by the section of the frequency spectrum that they alter. The following figures depict some of the basic types of digital filters:



High pass filters remove low frequency content.



Bandpass filters allow a section in the middle of the spectrum to remain.



Stopband filters remove a section in the middle of the spectum.

Lab 10: Implementing an FIR filter

In this lab, you will learn how to implement an FIR filter. The discrete convolution equation that describes the behavior of a causal length-N filter is:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

where x(n) represents the sampled input, h(n) represents the filter's impulse response (it's coefficients), and y(n) is the filtered output.

Part I: IO setup

ADC and Timers

In order to have a known sampling rate, the ADC must be interrupt driven. Set up your clock as usual, and let Timer B handle any filter clocks, if available. Set up the ADC to be interrupt driven and triggered by Timer A. Use the following code in order to set up Timer A: `/* TIMER A */ TACCR0 = 900; TACCTL1 = OUTMOD_4; TACTL |= TASSEL_2+MC_1;` Given these timing settings, what is the sampling rate?

DMA and DAC

Configure the DMA to transfer your filtered output sample to DAC0. You can set the DMA trigger inside the ADC12 interrupt service routine after you have finished the filtering.

Circular Buffer

Notice that FIR filter equation requires the N last input samples to compute the filtered output. In this case, you will need to store the last 21 samples including the most recent one. Implement a circular buffer to store all the required samples. Once you have reached the end of the buffer, reset your index back to zero.

Testing IO setup

Verify that the project is working so far by sending the newest value stored in your buffer to the DAC via DMA. Also, using the memory watch window, make sure that input samples are stored in their correct spot in the buffer. You may have to pause your program every time a sample is taken to verify that the buffer filling up correctly.

Part II: Filter Implementation

You will be implementing a 20-tap FIR filter. To obtain the filter's coefficients open up Matlab and type the following: `B = remez(20, [0,.5,.55,1],[0,0,1,1])` Next, plot the filter's frequency response with the `freqz` command. What kind of filter is this, and given the ADC sampling frequency, what is the expected cutoff frequency?

Notice that the array of coefficients includes negative numbers. They will have to be converted to Q-15 (16 bit), two's complement format. You can calculate the new coefficients on your own, or you may include the following line in your project: `signed short coeff[21]= {0x04F5,0xF4D0,0xFA12,0x03AF,0x0295,0xF86D,0xFCD1, 0x0D18,0x032C, 0xD768,0x3CC9,0xD768,0x032C,0x0D18,0xFCD1,0xF86D,0 x0295,0x03AF,0xFA12,0xF4D0,0x04F5};`

Inside your ADC12 interrupt service routine, compute the filtered output sample by taking the inner product of your input samples with the filter coefficients. The inner product is described mathematically in the equation above. Conceptually, your output can be described as `output_sample = most_recent_input_sample*coeff[0] + second_most_recent_input_sample*coeff[1]+...+oldes t_input_sample*coeff[20]`. This is a simple multiply and accumulate operation.

In order to get real time operation, the MSP's hardware multiplier must be used. Be sure to use signed multiplication. How can you use some of the other multiplier functions to simplify your code? How is the output of the multiplier stored? Since the DAC only transmits 16 bits at a time, which

multiplier register should be used as the output, and argue why this is a good scheme?

Part III: Testing

Hook up the function generator and the oscilloscope to the ADC and DAC of the board, respectively. Create a new coefficient array of length 21 consisting of a single one and the rest zeros. ([1,0,0,...,0]) What would you expect the output to be? Do you see that on the oscilloscope? Once this is working, try the original array of coefficients. Try changing the frequency on the function generator and make sure that filtering is taking place.

Now let'slook at the effects of the filter in the frequency domain. Set the oscilloscope to display the FFT of the input. The FFT, Fast Fourier Transform, will take a signal in the time domain and display it in the frequency domain. Look at the spectrum while using the [1,0,...,0] test coefficients and an arbitrary sine wave. Include a screenshot in your write up. Next, generate Gaussian White Noise using the function generator and connect it to the input of the board. White noise has power at all frequencies, so if we take the FFT of the output of the filter, we should get the frequency response of the filter. Does the actual frequency response look like the one generated by Matlab? Take a screenshot and submit it in the write up.

Part IV: Extra Credit

For extra credit, test your filter implementation with a new set of coefficients. Create a new set of filter coefficients that make up a new filter type. You may even increase the number of taps in the filter for more accurate results.

In order to generate a new set of coefficients, open up your favorite copy of matlab and use either the `remez` command. Newer versions of Matlab suggest you use `FIRPM`. If you are unsure how to use the command, type `help remez` to give you the syntax and usage. You may also try to use the Filter Design Toolbox for a more user friendly interface.

Once you have your coefficients, you must convert them to twos-complement Q-15 numbers. You may use twocomplement.m in order to do the conversion for you.